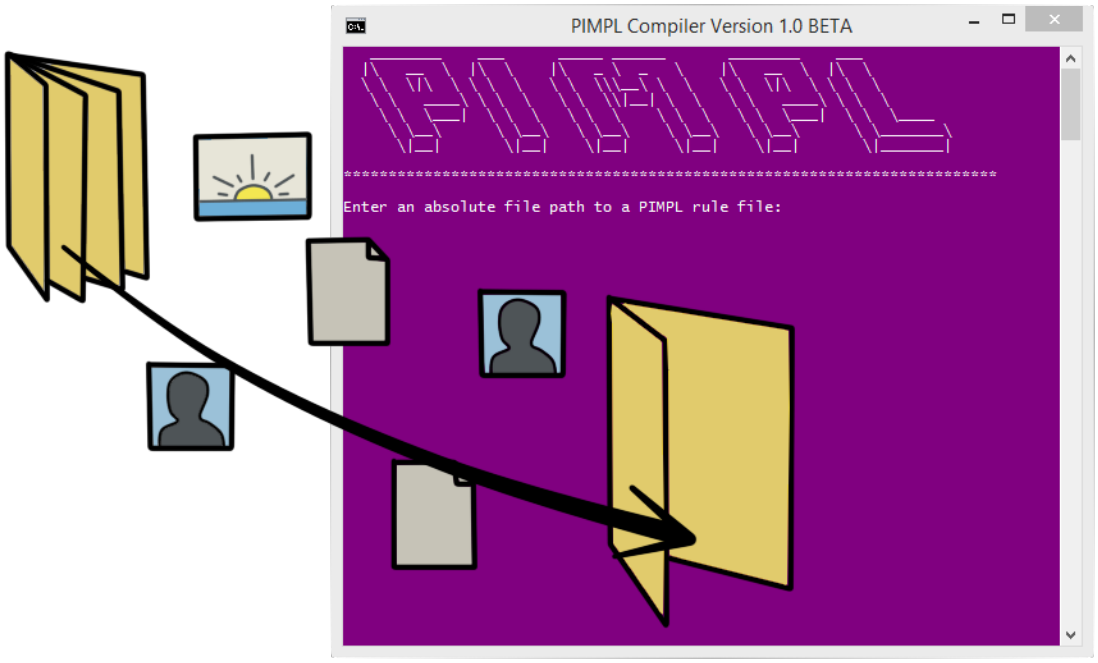
PIMPL

Organizing Files



Software P4 Project
Group SW405F14
Aalborg University
May 2014

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
Software
Selma Lagerlöfs Vej 300
9220 Aalborg Øst
www.cs.aau.dk

Title:

PIMPL: Organizing Files

Theme:

Design, Definition and Implementation of Programming Languages

Project:

4th semester

Project period:

February 2014 - May 2014

Project group:

SW405F14

Participants:

Jacob Elefsen
Patrick Grønhøj
Kenneth Haunstrup
Joakim Iversen
Martin Damgård Nielsen
Sofie Aaskov Nielsen

Supervisor:

Lone Leth Thomsen

Abstract:

The goal of this project is to support automation of file management with a new language and an accompanying language processor, for an assumed target group. Some existing solutions were examined and was found inadequate, especially with respects to metadata retrieval. The assumed target group includes, a primary and secondary group. The primary target group should have basic programming experience, while the secondary should have more advanced programming experience. The project resulted in a new programming language named PIMPL, which was designed and implemented with an accompanying compiler. The design of the language includes a structural operational semantic for a subset of the language. The project group conducted tests, which showed that programs written in PIMPL can be used for managing files in Microsoft Windows. The project group finds it likely that PIMPL would be useful for the assumed target group to manage files.

Editions: 8
Report pages: 157
Appendix pages: 49
Completed 28-05-2014

The content of the report is freely available, but publication (with source reference) may only take place in agreement with the authors.

Preface

This report is written by a group of 4th semester software students at Aalborg University. The course of the project, was commenced on the 3rd of February 2014, and the report was handed in on the 28th of May 2014. In the report the word “We” referees to the project group.

The group has chosen to create a programming language for *organizing files* in a Windows operating system. The group has received guidance from Lone Leth Thomsen.

Throughout the report terminology connected to the project is used. Therefore it is required that the reader has some knowledge about computer science, more specifically knowledge concerning programming languages, Java, C#, syntax & semantics and compilers.

The report uses Harvard citation, which first gives the name of the author and next the year of the source. For example [Hüttel, 2010], is the citation of Hans Hüttels book "Transitions and Trees". All the literature used is shown in the Bibliography section.

Appendix I, contains a CD and a list of content of the appended CD. The CD contains, among other things, the full PIMPL compiler source code and test programs.

Figures in the report is made with inspirations from figures in the books, [Hüttel, 2010], [Sebesta, 2013] and [Fischer et al., 2009], and is customized to the topic of the project.

Jacob Elefsen	Patrick Grønhøj
Kenneth Haunstrup	Joakim Iversen
Martin Damgård Nielsen	Sofie Aaskov Nielsen

Contents

1 Introduction **1**

2 Analysis **3**
2.1 Target group . . . 3
2.1.1 User needs . . . 3
2.1.2 Use cases . . . 4
2.1.3 Target group specification . . . 4
2.2 File systems . . . 5
2.2.1 Operations . . . 5
2.2.2 Metadata . . . 6
2.2.3 Folder traversal . . . 8
2.3 Existing solutions . . . 9
2.3.1 Languages . . . 9
2.3.2 Programs . . . 10
2.3.3 Summary . . . 11
2.4 Problem Definition . . . 11
2.4.1 Limitations . . . 11
2.4.2 Problem Statement . . . 12

3 Preliminary Choices **13**
3.1 PIMPL . . . 13
3.1.1 Language introduction . . . 13
3.2 Language Evaluation Criteria . . . 14
3.2.1 Criteria . . . 14
3.2.2 Priority and Trade-off . . . 16
3.3 Programming Paradigms . . . 18
3.3.1 Choice of Paradigms . . . 19
3.4 Language processor . . . 20
3.4.1 Key differences between compiler and interpreter . . . 22
3.4.2 Language processor choice . . . 22
3.5 Preliminary choices summary . . . 23

4 Design and syntax **25**
4.1 Syntax theory . . . 25
4.2 Comment . . . 26
4.3 Types and Values . . . 26
4.3.1 Variables . . . 27
4.3.2 Elementary types . . . 27
4.3.3 PIMPL specific types . . . 28
4.4 ThisFile . . . 30
4.5 Operators . . . 31

4.5.1 Precedence . . . 31
4.5.2 Coercion . . . 32
4.5.3 Elementary type operators . . . 33
4.5.4 PIMPL specific type operators . . . 33
4.6 Rule-part . . . 34
4.6.1 Options . . . 34
4.6.2 Rules . . . 37
4.7 Library-part . . . 39
4.7.1 General building structures . . . 40
4.7.2 Subprograms . . . 41
4.7.3 Selector . . . 42
4.7.4 Calculator . . . 43

5 Formal semantics **45**
5.1 Theory . . . 45
5.1.1 Structural operational semantics theory . . . 45
5.1.2 Type systems theory . . . 46
5.2 Big-step or small-step . . . 46
5.3 PIMPL's structural operational semantics . . . 46
5.3.1 Abstract syntax . . . 47
5.3.2 Environments . . . 49
5.3.3 Transition system . . . 53
5.4 PIMPL's Type system . . . 61
5.4.1 Abstract syntax . . . 61
5.4.2 Environment . . . 61
5.4.3 Type rules . . . 62

6 Implementation **65**
6.1 Preprocessor . . . 66
6.1.1 Procedure . . . 67
6.1.2 Grammar changes . . . 67
6.1.3 Line number mapping . . . 67
6.2 Syntax Analysis . . . 68
6.2.1 Theory . . . 68
6.2.2 Parsing strategy . . . 69
6.2.3 Handwritten or tool-generated syntax analysis . . . 71
6.2.4 Scanning and Parsing tool . . . 71
6.2.5 Transformation . . . 72
6.2.6 AST . . . 73
6.3 Contextual Analysis . . . 76
6.3.1 Choice of visitor pattern for the compiler . . . 77
6.3.2 Type checking . . . 77
6.3.3 Scope rules . . . 78
6.3.4 Decorated AST . . . 81
6.4 Code generation . . . 81
6.4.1 PIMPL runtime . . . 82

6.4.2 Implicit variables . . . 83
6.4.3 Persistent variables . . . 84
6.4.4 Expressions . . . 85
6.4.5 Code emission . . . 85
6.5 Error handling . . . 86
6.5.1 Error handling during compliation . . . 86
6.5.2 Runtime errors . . . 86
6.6 Testing . . . 87
6.6.1 Testing during development . . . 87
6.6.2 Full compiler test . . . 87
6.6.3 Extended testing . . . 92

7 Discussion **93**
7.1 PIMPL language evaluation . . . 93
7.1.1 PIMPL in relation to the existing solutions . . . 93
7.1.2 PIMPL limitations . . . 94
7.1.3 Paradigm choice . . . 95
7.1.4 Language evaluation criteria . . . 95
7.2 Method evaluation . . . 97

8 Conclusion **99**

9 Further Development **101**

Bibliography **103**

Appendix **106**

A Existing solution test **107**

B Readable EBNF Grammar **111**

C Implementation LL(1) Grammar **115**

D Lexical elements **121**

E Design and Syntax for elementary types **125**

F Example of a small PIMPL program **129**

G Structural Operational Semantics **133**

H PimplLib class diagram **155**

I CD **157**

1 Introduction

The goal of this project is to support automation of file organization with a new language and an accompanying language processor.

Handling multiple files using a graphical user interface in the shell of the modern operating systems can be tedious and repetitive and people tend to find it frustrating not to be able to retrieve their personal information when needed [Richard Boardman, 2004].

Searching for files using a tool that indexes files in the file system might be useful but this requires some degree of recall to remember what the file is called as opposed to recognizing a folder with a given name. A study indicates that users actually prefer folder-based navigation rather than searching for specific names [Barreau and Nardi, 1995].

A user will have to make archiving decisions and move files manually every time a new file has entered the file system. These files could for instance be downloaded from the Internet or transferred from a camera. Many tasks, like organizing all files in a designated download folder by file type, could be automated by different kinds of existing software and programming languages.

This project is subject to some requirements, according to the study regulations of this semester. The project must result in a new programming language and a language processor for that language, which is the primary focus of the project. The benefits of creating a programming language as opposed to an application, is a more general and flexible solution. A programming language can allow the users to implement their own solution to their individual problems, because they are not limited by what other programmers have programmed into an application.

An analysis of the problem domain, a design for a new language and description of the implementation of a language processor for this new language is presented throughout the report.

Analysis 2

This chapter presents a target group for which such a language could be interesting and some accompanying use cases. Later an introduction to file systems, and some existing solutions associated with the subject. The above-mentioned is analyzed in order to get an idea of how a new language could support its target group.

2.1 Target group

This section describes the target group for the new programming language. This programming language has a specific type of task to solve but does not have a single obvious target group. This section contains an analysis of potential user needs, then some examples of use cases will be described and lastly the target group will be specified. The project will not include interaction with potential users, because this is not the focus of the project.

2.1.1 User needs

Studies show, according to [Barreau and Nardi, 1995], that computer users consciously organize their files for easy retrieval. Although a questionnaire executed by Danmarks statistik indicates that 8% are well aware of their inability to organize their files in a way so that they can easily retrieve them again and that 12% expresses limited capabilities of organizing files[statistik, 2009]. 20% of the questioned says that they need some way to get their files organized. This study shows that there is a need for file organization.

How do the users access their files? As mentioned in section 1, the users tend to prefer "folder based navigation" [William Jones, 2014; Barreau and Nardi, 1995] as opposed to performing search queries for files. This is because search queries requires a high degree of recall, since the user would have to recall some detail, like a name, about the file the user is looking for. Whereas recognition of folder structures, where the user would have to recognize steps of folders to traverse in order to get to the desired file, does not require any recall, only the recognition of the folder names or structure. As the amount of storage space and files increases, it gets difficult to keep track of personal files [Shankland, 2013].

Users might not feel that their time is spend optimal organizing ones' personal files, they might save time if at least some parts of the work could be automated [Patrick Darling, 2010]. To sum up, users prefer files to be organized in folders over searching for files and some find it troublesome to keep track of where they have archived their files. A new programming language could help the users create programs tailored to their needs, which assist in automating file organization tasks.

2.1.2 Use cases

There are a number of situations where it could be helpful with a programming language to help organizing files on a computer. These use cases are based on the project groups own experience, as no target users have been questioned. Some examples of concrete use cases for file management are presented in this section. These use cases show examples of what needs people have and how these can be helped.

Organizing pictures

File organizing is useful when people transfer pictures, captured by their digital cameras, to their computers. The pictures taken would typically come with some sort of auto generated name, a name without meaning in the context which do not make sense as a basis for sorting the pictures [Wikipedia, 2014a]. One could, for instance, be interested for a way to split one's pictures into folders based on the context the pictures were taken in, or to rename the pictures with more meaningful names. The context could for instance be derived from the metadata stored in the images files, such as: GPS data, the date the picture was captured, from what camera the picture was captured, and other image related metadata. An example where sorting by metadata could be useful could be a family who just came home from a vacation and have captured a lot of pictures, these pictures are important moments and is therefore be stored for later retrieval. This shows that a certain need for organizing files is needed [Richard Boardman, 2004].

Storage capacity issues

Another use case is when there is storage capacity issues on computers. The number of files stored in personal computers are increasing, which is described in section 2.1.1. At the same time more compact computing units such as Netbooks, Ultrabooks, and tablet/laptop hybrids, are also becoming increasingly popular [Mikey Campbell, 2013], even through the compact format puts severe constraints on the total amount of available storage. Over the years more and more pictures are being taken and saved onto computer hard drives. According to Instagram, around 60 million pictures are being shared each day and 20 billion since they first launched back in late 2010 [Instagram, 2014]. This indicates that there is a lot of files in circulation.

A download folder often contains duplicates or unused files. In this case it would be advantageous to have a programming language to set up rules to remove duplicates and remove old files, for instance with extra rules for file sizes and file types.

Summary

Both of these use cases shows examples of situations, where better organization of files could improve the user file management experience.

2.1.3 Target group specification

The project group has chosen to focus on creating a new language for private users, although a new programming language which helps overcoming the problem might be useful for a wide variety of users.

There are two target groups, a primary and secondary, for the new programming language. The primary target group should have basic programming experience, while the secondary target group should have more advanced programming experience.

When we reference to "Novice user" and "Experienced user" throughout the report, we are talking about the primary target group and the secondary target group.

2.2 File systems

This section include a description of file systems in the Windows 7 operating system, as seen from the users aspects. The file system is examined to get an understanding of the problem domain of the project. File systems are overall structured with folders to categorize files. There exist both files- and folder operations available for e.g. move, copy, rename and delete, so that the file system can be kept organized.

Windows is chosen as the target platform because it is the most widely used operating system compared to e.g. Mac or Linux[W3schools, 2014].

There is more data to a file than the content of the file itself, which is the case with Windows. There is typically additional data about the file like how the file should be read by a program, and data addressed to the user of the file system, for instance about who created the file. This data is called metadata or "data data".

This section focuses on the metadata which is addressed to users. This data is information which could be used to categorize and search for files in the file system. There are descriptions of file operations, metadata and how to traverse folders in the following sections.

2.2.1 Operations

This subsection focuses on the operations which can be done on files and folders. The operations that are relevant for the project are the ones concerning file organizing, e.g. moving, copy, deleting and renaming files or folders which must be done to ensure order in the users storage [Cem Ozdogan, 2011].

The relevant operations on files and folders are listed below. Operations like creating a file, editing and open/close files are not included, because they are not needed in order to organizing files [Alverno College, 2012].

- **Deleting a file or folder**: When deleting a file or folder, the file is removed for the directory and moved to the recycle bin. In the folder, all the content are also deleted.
- **Renaming a file or folder**: Changing the name of a file or folder.
- **Copying a file or folder**: Creates a clone of a file or folder in the specified place. When copying folders all the content is also copied.
- **Moving a file or folder**: To move a file or folder means, moving the specified file or folder to another place. Like when copying folders, all the content is moved.

There are some extra operation for folders, which is to create a folder or include/remove it in/from a library.

- **Creating a folder**: This makes a new folder in a given place.
- **Include in library**: When a folder is included in a library, it is not moved or copied to it, instead the library gets a reference to the folder.
- **Remove location from library**: The folder is being removed from a library. This does not delete the folder, but removes the reference to that folder.

2.2.2 Metadata

This section focuses on some of the file type classes that are used frequently among private users. Microsoft Windows 7 uses four standard folder libraries, with focus on one file type class each. These libraries targets document-, image-, audio- and video files [Microsoft, 2014a]. It is therefore assumed that these file type classes are used frequently by private users. A file organization language needs the opportunity to work with these file metadata.

Each of these, and their associated specified metadata, will be described in the following subsections. Metadata can be saved as a part of a file, or it can be placed in the file system. As an example a JPEG file might contain a header with metadata about what camera model was used to capture the picture, a thumbnail to preview the image and then the primary data, such as the image itself [Stock Artists Alliance, 2014]. See figure 2.1 for a visual representation of a file.

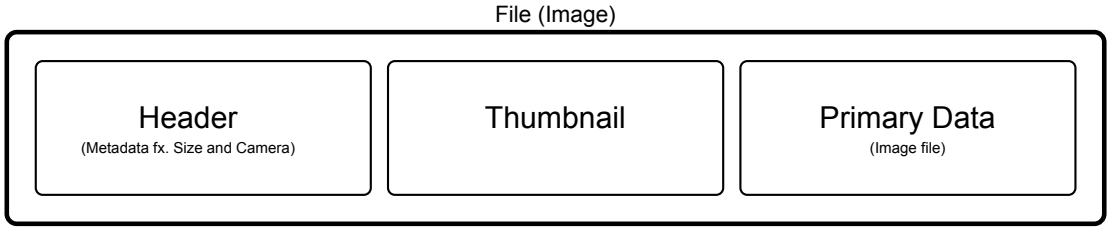


Figure 2.1. Illustration of metadata in a jpeg file.

Metadata can be used to organize files, and can be used to search for files. Many files can be categorized by content the content of the files' metadata, this could be categorization by author name or the likes. The following lists of metadata is derived from the details shown about files in Windows 7, although more data exist for each file.

A file does not necessarily contain data for each of its possible fields. For example some cameras do not save GPS location while others do. When there is not saved any data, the field is just empty when the details is showed about a file.

All files in a Windows operating systems includes certain metadata, all categorized under File. These are referred to as the standard file metadata:

- **File:** Name, Type, Folder path, Size, Date created, Date modified, Attributes, Owner, Computer.

For the file types focused in this project, the files have more metadata in common. Both the Description and Origin categories are in image, audio and video files, these categories has the following fields:

- **Description:** Title, Subtitle, Rating, Comments.
- **Origin:** Copyright.

In addition to the above-mentioned metadata, some file types have some speciel metadata. In the following subsections there will be described some of these metadata for image, audio and video files.

Image files

There are a number of different file types to images, and they can have a number of different metadata. This subsection is limited to only one file type, the JPEG file type. This file type is chosen, because most digital cameras save images in JPEG [Wikipedia, 2014g]. The user addressed metadata, which is special for the JPEG files, is listed below. Note that not all JPEG files have GPS data, so this item is not always in the details about a JPEG file.

- **Image**: Images ID, Dimensions, Width, Height, Horizontal resolution, Vertical resolution, Bit depth, Compression, Resolution unit, Color representation, Compressed bits/pixel.
- **Camera:** Camera maker, Camera model, F-stop, Exposure time, ISO speed, Exposure bias, Focal length, Max aperture, Metering mode, Subject distance, Flash mode, Flash energy, 35mm focal length.
- **Advanced Photo:** Lens maker, Lens model, Flash maker, Flashmodel, Camera serial number, Contrast, Brightness, Light source, Exposure program, Saturation, Sharpness, White balance, Photometric interpretation, Digital zoom, EXIF version.
- **GPS:** Latitude, Longitude, Altitude.

In addition to this metadata, there are more fields in the categories Description and Origin than mentioned earlier. Image specific metadata is listed here.

- **Description:** Tags.
- **Origin:** Authors, Date taken, Program name, Date acquired.

Audio files

There exist many different audio file types, but the metadata for these are roughly the same, which means the project will not focus on some specific files, but more the overall class.

Examples of what kind of metadata that can be stored with audio files:

- **Media:** Contributing artist, Album artist, Album, Year, #, Genre, Length.
- **Audio:** Bit rate.
- **Content:** Parental rating reason, Composers, Conductors, Group description, Mood, Part of set, Initial key, Beats-per-minute, Protected and Part of a compilation.

Like with image files there is more metadata fields in the category Origin. These are listed below.

- **Origin:** Publisher, Encoded by, Author URL.

Video files

As with image and audio files, many different video extensions exists. The metadata for all these files are very similar, and most often they are identical.

- **Video:** Length, Frame width, Frame height, Data rate, Total bitrate, Frame rate.
- **Audio:** Bit rate, Channels, Audio sample rate.
- **Media:** Contributing artist, Year, Genre.
- **Content:** Parental rating, Parental rating reason, Composers, Conductors, Period, Mood, Part of set, Initial key, Beats-per-minute, Protected.

Video files has also some extra metadata in the categories Description and Origin.

- **Description:** Tags.
- **Origin:** Directors, Producers, Writers, Publisher, Content provider, Media created, Encoded by, Author URL, Promotion URL.

2.2.3 Folder traversal

This section contains an overview of how to organize files, and how to make them easy to find again. It is necessary to know how the files should be found, before the files can be organized well. This is also needed when designing a language to help organize files, because it should provide the necessary constructs to make this kind of organization.

Therefore this section covers a description of how computer users want to search for files. Afterwards the section covers how the files may be organized.

When users want to find a specific file or folder, there are three main approaches: query searching, traverse folders or with the use of libraries. Searching for a file or folder requires the user to remember the name, or other significant data about the file or folder, to find it in. Windows 7 has the possibility to use filters, which allows the user to not only search by a name, but also for the file type or metadata. It is also possible to select a specified folder to search in. To traverse folders it is only necessary to recognise names and structures, and not recall the exact name or file type for example [Microsoft, 2014a]. The last approach, libraries, is a relative new thing in Windows, where files can be included no matter where the files actually are placed, for easy retrieval and categorization. The same file can be placed in multiple libraries at once. Some standard libraries are Pictures, Documents, Music and Videos [Microsoft, 2014b].

As mentioned in section 2.1.1, users prefer folder traversal rather than searching, but to find files with libraries works a bit the same way as finding normal folders. Therefore the rest of the section will focus on finding file and folder by traversing or by using libraries.

There are many different ways to organize files, and none can be said to be the best one, because it often is up to the user how to sort their files. A good way to organize files depends on which type of files, how many, and in which way they need to be used. A possible way is to look at the metadata about the files, and base the folder structure on it. For example images can be placed in a folder named “Images”, with subfolders for each year, and further subfolders with events where the images are from. A concrete image could be placed like this:

Images − > 2012 − > *My birthday* − > *Concrete image*

When it is decided how the files should be organized, there is a need to place all the files in the new order. To do this the user can for example create all the wanted folders, and then one-by-one, check where the file belong and then move the files to the decided place. It is also a possibility to use other file- and folder operations, like renaming and delete, to make a good structure. The user can also submit a file or folder to a library, but this also requires to look on every file or folder, and decide where it should go.

A language should provide this functionality, but make it easier work on more files at the time.

2.3 Existing solutions

There already exists different programming languages and programs, which gives the possibility to make file operations on computers. In this project it is limited to the computers running a Windows operation system, as mentioned in 2.2. The languages and programs work in different ways, and they offer different opportunities in the way the user can organize their files. Appendix A presents tests performed on each of the existing solutions in order to see if they can complete the staged tasks. The new programming language will also be run through these test, to see if is a better solution, see section 6.6.2 for the language tests.

A few existing languages and programs are examined in the following sections. To examine the programs and languages, and to see how well these can be used to organize files, these solutions are used to solve some specific problems. The problems are created from the use cases, and can be seen in appendix A, which also shows the problems and the results from each solution.

The first language, Java, is mentioned because of its popularity [Tiobe, 2014], and PowerShell is mentioned because it is a tool for system administrators, provided by Microsoft, with the purpose to be used to automate tasks such as file management. The programs presented are the two first entries from a Google search for "automated file manager", it is assumed that the algorithm used by Google to rank search hits by relevance finds two popular tools. Section 7.1.1, contains a discussion of the existing solutions and the new programming language developed in this project.

2.3.1 Languages

This section contains a description of the two languages Java and PowerShellScrips.

Java

Java is a general purpose programming language (GPL) [Oracle, 2014] that includes many features. It is intended for programmers to "Write once, run anywhere", meaning it is not needed to be compiled more than once to run on a different platform. Java is widely used around the world, for example 89% of the desktop computers in the U.S. runs Java [Java, 2014].

Java is a general language and Java is able to utilize many features on a given platform[Java, 2014]. A user facing a file organizing problem could indeed write a program in a GPL like Java that solves the problem. But Java does not contain specific language constructs supporting file organization. A program in Java needed approximately 130 lines of code to solve the tasks set in appendix A.

PowerShell Scripts

PowerShell is a Microsoft framework for task automation and configuration management. It uses a language called PowerShell script (PS-script). PS-scripts allows easy access to a lot of the features in a Windows operating system and the accompanying software packages. Administrators can use PS-scripts to automate and control the administration of a Windows operation system[Microsoft, 2014d]. PowerShell can be used to write scripts that organize files. But more advanced features like using specific metadata from a picture is relatively difficult to do. The PowerShell test shown in appendix A showed that it is difficult to write file organizing programs in PowerShell.

The PowerShell environment contains a rich set of features and many operations are already present like the command "Clear-Content" which is used to clear the content of a file [Microsoft, 2014e]. The same properties that make it easy to write powerful scripts with PowerShell might also make it harder to understand what a script does because very complex actions hide under only a few statements.

2.3.2 Programs

There exists some programs and tools that can help with organizing files for instance Belvedere and DropIt, these are described in this section. Both programs allow the user to clean-up and organize folders in Microsoft Windows. These are tools which can e.g. move, copy, delete and rename, based on file attributes such as: name, size and creation date and the likes based on rules specified by a user. In both programs the user sets up rules, which is then enforced on the targeted folders. Neither of them have the possibility to delete duplicates, the closest is to delete files with a given sub-name like "- Kopi" [Pash, 2008] [Lupo PenSuite Team, 2014].

Belvedere has an easy and intuitive user interface for simple folder selection and rule creation. The program lacks some features like the ability to access and use metadata in different ways.

As an example, the task could be to select all files from a given year, though it is only possible to select files from the last x weeks, or to access file specific metadata, like when an image is captured. In Belvedere there are only a fixed set of available actions, and just a few ways to identify files. A user would for instance not be able to create rules for renaming images to their taken date. Another example can be made for sorting image files. Belvedere can only identify these files by the basic file metadata, see section 2.2.2. When looking at image files, users might often look at the metadata specific for the file type for categorizing these, but this is not possible when using Belvedere [Pash, 2008].

DropIt works a bit different, but the basics are the same. DropIt is a mix between 'drag and drop' and menus in a GUI and some code writing. The GUI part has some limitations, but

the code part gives more opportunities to the way the user can organize files. However the combination of both code and GUI makes it a bit confusing to use. Some things can only be done in the GUI, some things only in code and some in both. DropIt allows users to rename files and folders based on the metadata specific to their types, but it still lacks the opportunity to work exclusively with files with specific metadata [Lupo PenSuite Team, 2014].

In summary, both programs can be used to organize files automatically according to rules set by users. More advanced categorizing operations are not available, so the users has to make compromises when organizing files.

2.3.3 Summary

The existing solutions all seems to agree on a certain feature set for file managing including: copying, deletion, moving files, folder creation, renaming files and folders. What they do not agree on is how to specify the selection of files and how to specify what operations should be performed.

DropIt includes limited ability to read and use metadata, only the standard file metadata is available. Belvedere is even less capable of utilizing the available metadata. Although programs written in Java and scripts written in PowerShell are both capable of using metadata specific to certain filetypes, though it is very complicated and requires multiple constructs and import of special libraries just to get e.g. the creation date of a given file. But full support for metadata in a file managing language might be useful in many contexts: Selection of files and deleting these, for instance: Select all files created yesterday and move these to the recycle bin. Categorizing of files, for instance: Move all picture files taken in Denmark to a new folder.

The new language should be able to access file type specific metadata with possibilities similar to those found in GPLs such as Java, but by writing less code and at the same time still be readable for the target group.

2.4 Problem Definition

In this chapter, the target group, file system and existing solutions have been described. The target group has been specified to be novice and experienced programmers. The graphical user interface to the file systems used in a Windows operating systems has been examined and this has resulted in an overview of potential functionality, which could be included in the new language.

The examined solutions have some limitations related to the user needs. The general purpose programming languages(GPL) that have been examined provides the facilities required to create solutions for the evaluation task in appendix A. A GPL is quite complex and not specific for the domain. The programs which have been examined do not include all of the desired functionalities and customization options.

2.4.1 Limitations

The time for this project is limited to the duration of a semester (4 months). The project only addresses the way files are handled in Windows.

The project group has chosen not to focus on the library feature in Windows. These libraries were activated by default in Windows 7, and in Windows 8.1 these are disabled by default and requires a few changes to enable. This could indicate that these libraries were not used and so have been turned off.

2.4.2 Problem Statement

Based on the problem definition and the limitations from the previous sections, the project will focus on the following:

The purpose of the project is to design a programming language and implement an accompanying language processor which can provide the target group a way to automate file management in the Windows operating system.

The following questions has been attempted answered:

- For the programming language
    1. How can a programming language make it possible to run file operations on files, to automatically organize those files once or recurring?
    2. How can a programming language give access to features like the gathering of metadata, and the use of these to organize files?
    3. How can a programming language make it possible to run file operations on several files without the user to do it for one file at the time?
    4. How can a syntax be constructed for such a programming language?
    5. How can an operational semantics be constructed for such a language?
- For the language processor
    1. How can a language processor be implemented to process program code in the programming language?

Preliminary Choices 3

The preliminary choices of the project are presented in this chapter. These preliminary choices are used as a basis to define and design the new language. At first some Language Evaluation Criteria are presented and discussed, next some major programming paradigms are presented and a set of paradigms are chosen. After these sections, the different types of language processor types are discussed, in order to chose a language processor type.

The new language presented in this report is given a name in order to help make a discussion about it and presentation of it. This chapter therefore includes an introduction to the language.

For all of these preliminary choices, some different options were analyzed, and based on this analysis, a choice was made.

3.1 PIMPL

The project group has chosen to name the new programming language PIMPL. PIMPL is an abbreviation for **P**ersonal **I**nformation **M**anagement **P**rogramming **L**anguage. Information Management is a term used for the management of information needed in order to enable an optimized access to it [Wikipedia, 2014d]. We found that this was an appropriate term to use and we added Personal, because the language is targeting private users of personal computers. Programming Language was added to clearly state, that this is a programming language.

3.1.1 Language introduction

The goal of PIMPL, as described in the problem definition in section 2.4, is to organize files. Another goal is to make programming file organization programs easier for the target group, and this should result in a language that is highly abstract. This for example means that the users does not need to design algorithms for traversing folders and locating files. The traversal of folders are always done in the same way, because the program has to visit all files in order to check if they should be selected. It is therefore unnecessary for the user to design and write the traversal algorithm. When files are selected, file operations can be made on these.

The traversal algorithm should be hidden, as this could potentially make it easier for novice users, as they do not need to worry about this algorithm and cannot make changes to the algorithm, that might cause errors in the traversal of folders.

It is possible to define how a program written in PIMPL should be executed. A PIMPL program can be configured to run once or recurrently. This option is available in order for the users to make programs that automates the file system. With recurring it is possible to maintain the desired folder orders automatically, where once runs the program once and then terminates.

A PIMPL program is divided into two parts; a Rule-part and a Library-part. These parts must be divided into different files, in order to organize program code and to share Library-files. The main part of PIMPL is the Rule-part, which is targeted at novice programmers. The second part is the Library-part, which is targeting the experienced users.

The Rule-part consists of, the primary part of the program, such as defining what folders to work on, the selection of files and the action on these. The Rule-part will be described in detail in section 4.6, but in short, it defines some rules with a **Select** to chose files, using **Selectors**, and a **Do** to make operations on these files. The Library-part consists of different subprogram types, called **Selectors** and **Calculators**, which are used to select which files the user wants to work on. The Library-part will be further described in section 4.7. A user manual has been made as a guide for the users, this user manual can be seen in appendix I.

3.2 Language Evaluation Criteria

When creating and evaluating a programming language, criteria can be used in order to assure the quality of the programming language. The priority of the criteria is made to help design and build a programming language that fits the PIMPL target groups. These criteria are described and discussed in this section. A discussion of the trade-offs follows, as well a table with the priority of the criteria is presented.

3.2.1 Criteria

There exist many possible language criteria, but some of the most commonly agreed criteria have been chosen for this project [Sebesta, 2013]. Beyond these commonly agreed criteria, the project group has chosen some, which seems relevant to discuss in relation to the design of PIMPL. These chosen criteria includes readability, writability, reliability, learnability, implementability, orthogonality, efficiency, generality, portability and maintainability.

All of these criteria influence each other in different ways. Sometimes the criteria can overlap and sometimes they are contradicting. It is different from problem to problem, which criteria the language designer should focus on. This section will describe each of these different criteria.

Readability

Readability is a way to describe how user-friendly a programming language is to read and understand. Readability is one of the core aspects of a programming language when it comes to code maintenance. Readability decreases the required time needed

for understanding programs written in the programming language. Simplicity in a programming language increases the readability. A programming language that allows many ways to accomplish an operation is bound to be less readable, language features such as operator overloading can also make the code less readable if the overloading is not intuitive [Sebesta, 2013].

Writability

Many of the same concepts that make a programming language readable play a big role in the writability of the programming language. Good writability means that the language is convenient, rather than cumbersome. A short form of a keyword can for instance make the writability of the language better, for example if the user can write "f" instead of "float". But writing a keyword in shorter form or even having multiple ways of writing the same keyword can greatly reduce readability. Having operators with a lot of functionality can also greatly increase the writability of the programming language [Sebesta, 2013].

Reliability

Reliability in the context of a programming language means that it has to perform to the specifications set under all conditions. Some language features have an impact on the reliability of a program. One of these features is type checking. Type checking is testing for type errors in a written program during compile-time or during program execution. Type checking is an important part of reliability in languages. The sooner a type error is discovered, the less expensive the required repairs will be [Sebesta, 2013].
As mentioned, readability and writability have an important effect on how a program is written. If a language has low readability and writability then it is more likely to be less reliable [Sebesta, 2013].

Learnability

Learnability is about how easy it is to learn the language. Where easy can be understood as how quickly users understands the language, and how fast they can make the desired programs in the language. Learnability correlates well with the readability criteria.

Implementability

Implementability is about how demanding and time consuming it is to implement a language processor for the language. Implementability is also about the ease of creating a language processor. High implementability means that it is relatively fast and easy to implement a language processor.

Orthogonality

Orthogonality generally means that the programming language is consistent and simple. A more technical description is that the programming language only contains a small set of constructs and these can be combined in a small number of ways to build the control and data structures. Furthermore, every possible combination of these primitive constructs is legal and meaningful [Sebesta, 2013].

Efficiency

Efficiency is about how much CPU power and memory a program written in a given language requires in order to run. In highly efficient languages, efficient programs can be written, that uses as little as possible CPU power and memory.

Generality

Generality in a language is about combining closely related constructs, in order to form a standard way to make a certain operation. This could e.g. be about having only one way to make a loop, where as some languages offer two or three ways to make a loop. Generality is also about having general language constructs that makes a language usable in a wide variety of tasks.

Portability

Portability defines if the language should be able to run on different platforms or machines. A high amount of portability means that the language is universal and is able to be used on many different kinds of platforms.

Maintainability

Maintainability is the way to describe the difficulty of correcting errors, fixing issues or adding new features to a language. In order for a language to be maintainable, a very strict coding style and good categorization and adequate documentation is needed. This can support writability.

3.2.2 Priority and Trade-off

The criteria prioritized ranking from Very Important to Not important. Very Important and important prioritized criteria will be considered when PIMPL is designed and created, in an attempt to accommodate these criteria. Less important and Not important are at first not considered, but is not necessarily excluded from consideration. At the end of the project, PIMPL is evaluated based on the criteria set and the priority of these. The evaluation of the language, and the chosen criteria, can be seen in section 7.1.4.

This section contains the criteria choices made for the language. A prioritized list has been made and is shown in table 3.1.

Criteria	Very important	Important	Less important	Not important
Readability	✓			
Writability		✓		
Reliability		✓		
Learnability	✓			
Implementability	✓			
Orthogonality			✓	
Efficiency			✓	
Generality				✓
Portability				✓
Maintainability				✓

Table 3.1. Priority of criteria in the project.

Readability is prioritized as Very important, because the language should be as easy as possible for the primary target group. In the design of PIMPL, it is important to make choices that makes the language more readable. These choices could be e.g. to choose meaningful keywords, simple language constructs and to write keywords using pascal case for easier recognition.

Writability is categorized as Important, because the project groups wants that programs in the language are easy and fast to write. It is not set at Very Important, because some of the features that make a language writable have a negative impact on the readability of the program. Readability is Very Important, in order to avoid it becoming difficult for the users to understand and write the desired programs in a relatively fast way. It is not recommended to use long keywords.

Reliability is prioritized as Important, because written programs must do exactly what the user wants. When making these file operations, small errors can remove important files and thereby give major problems to the users.

Learnability is closely related to readability, and is also set to Very important for the programming language criteria. Here it is important to consider choices that make PIMPL simple and with relatively few, meaningful and understandable constructions.

Implementability is also Very important as the project needs to include an implementation of a language processor, which is able to process PIMPL. It is also the project groups first language processor, and there is a limited time frame for implementing this, this means that it must be relatively simple to make an implementation. It is therefore not desirable to include complex language constructs which would take a long time to implement in a language processor.

Orthogonality is categorized as Less important. The project groups thinks that a language is easier to learn and remember if there are similar rules for writing expressions, statements or combinations of these.

Efficiency of the language is categorized as Less important because file operations are slow in general, as hard drives are the bottleneck of modern personal computers' file operations.

Generality is categorized as Not important, and this is because PIMPL aims at being a language targeting a specific domain in programming.

Portability concerns with the language being portable, meaning it is easy to make it usable on platforms other than the targeted Windows platform. This is not important for the project, due to time limitations and the project focus on making one language processor.

Maintainability is set as Not important for this project, as the language will not be publicly released and only used for the project evaluation. It is only necessary to maintain the language in the limited development process.

To sum up these choices, the primary focus is the readability aspect of the language. This is supported by learnability for easy learning and implementability for writing an associated language processor. Other than these, aspects such as writability and reliabiliy is being taken into account when designing PIMPL. Orthogonality and efficiency as well as generality, portability and maintainability will not be the weighted much when designing PIMPL.

3.3 Programming Paradigms

A programming paradigm describes the style and capabilities of a programming language. Programming paradigms can be used to classify a programming language in order to enable a quick general understanding of a given programming language. Specifying a programming paradigm for a language helps understanding the general idea of the language and provides a rough idea of what it should look like. The design of the new language is based on the chosen paradigm.

There are four main paradigms, these are the: imperative-, functional-, declarative- and object-oriented paradigms [Sebesta, 2013]. Some languages are build around just one paradigm, but most languages like Java, C++ and C# contain elements from multiple paradigms. The four main paradigms are introduced in this section in order to find out which possibilities exist in order to discuss which paradigm or combinations of paradigms PIMPL should be designed on the basis of.

Imperative

An imperative programming language includes statements that can be used to define and manipulate state. An imperative language tells the computer exactly what to do step by step, statement by statement, and directly changes the state of a program. The imperative paradigm has been phrased "First do this and next do that" [Nørmark, 2013].

The imperative paradigm is directly derived from modern computer architecture [Wikipedia, 2014c], which processes statements in sequence. Imperative languages handle control flow with conditional statements, loops and sequence. Examples of imperative languages are ALGOL, FORTRAN and C [Wikipedia, 2014c].

Functional

Functional languages are based on mathematical functions, and does not include variables. A mathematical function describes a mapping between parameters and return values. Unlike in imperative languages where control flow is managed with conditional statements, loops and sequence, functional languages makes use of conditional statements and recursive structures, because there are no variables to use for instance with counting loops. Functional languages do not include changes in states as its known from imperative languages, this guarentees that functions cannot have side effects [Sebesta, 2013].

Functional languages can be seen to have more readability than their imperative counterparts, because expressions and functions do not have side effects, and users can read specific isolated parts of a program i.e. one or more functions, and understand what that specific part does, without reading the whole program, because a function always does the same. Examples of functional programming languages are LISP, F# and Haskell [Sebesta, 2013; Wikipedia, 2014b].

Declarative

Declarative languages, are rule-based and non-procedural, unlike imperative and functional languages. A declarative language describes how the results should look like, and not how to get to the result. It does not give orders directly to the computer, but instead makes the computer find a correct answer. The declarative syntax and semantics are remarkably different from that of the imperative and functional language, according to [Sebesta, 2013].

Declarative languages differ from imperative and functional languages in which there are no assignments and no control of the flow of the program. Examples of declarative programming languages are PROLOG and SQL [Wikipedia, 2014e].

Object-oriented

The Object-oriented paradigm is generally about modeling concepts using objects. Object-oriented software systems allow highly modular designs with good reusability. The object-oriented paradigm has good support for encapsulation and logical grouping of program elements.

Examples of object-oriented languages are Smalltalk, Java and C# [Wikipedia, 2014f].

3.3.1 Choice of Paradigms

The existing program solutions described in 2.3.2 all make use of rules to specify how files should be organized, but the users do not need to specify algorithms, only how files should be handled when encountered. This might suggest the use of a purely declarative or rule based programming language where the traversal algorithms are never specified. Although the language cannot be declarative, as some form of sequence is required. As an example, if the users define two Rules, where the first Rule moves certain files from a folder and the next Rule deletes the remaining content of that folder. If there is no way

default way of handling this, then program behavior can change from time to time. This is not a desired effect, and this means that the declarative paradigm is not used.

If the declarative approach was used, it would result in an entirely different language. Though it would not be possible to work on the same files in different Rules, as there is no way to ensure that one Rule runs before another. The users must make sure not to make Rules that can overlap one another. All this indicates that an imperative paradigm could be a solution to this problem.

The imperative paradigm has been chosen, as a result to the problems with the declarative paradigm. The imperative paradigm uses sequence to control the flow of the program.

Overall a single algorithm for traversing folders seems to be all that is necessary in order to locate files. More specific algorithms to chose files are needed for the selection of files. Organizing files never include putting a file in a relative order to other files, it is only about placing files in intended folders. The algorithm used is then executed, each time it encounters a new file, and takes the action according to the Rules specified in the program. This could indicate that a language with a high level of abstraction could be helpful for the novice programmers.

However PIMPL's Library-part, needs to include capabilities to specify algorithms, which describe how to choose files. The Library-part is also chosen to be imperative, but with a lower abstraction level. Since the language targets a specific domain, namely file management, PIMPL should be considered an imperative domain-specific language (DSL). The following quote explains the characteristics of a DSL:

> "DSLs trade generality for expressiveness in a limited domain. By providing notations and constructs tailored toward a particular application domain, they offer substantial gains in expressiveness and ease of use compared with GPLs for the domain in question, with corresponding gains in productivity and reduced maintenance costs" [Marjan Mernik and Jan Heering and Anthony M. Sloane, 2005].

3.4 Language processor

This section contains a short description of what compilers and interpreters are, two different ways of processing a program in a language, and what they do.

There exists compromises between compilers and interpreters called hybrid implementation systems. These hybrid systems translate a high-level language to a lower-level intermediate language which is designed to be easily interpreted [Sebesta, 2013].

The goal with this section is to give an understanding of the differences between the two main types of language processors. The language processor types are presented here in order to facilitate a discussion about which type of language processor should be implemented for PIMPL.

Compiler

A compiler is a program, which translates i.e. compiles a program written in some programming language into a program written in some other programming language or into machine code. It is often from a high-level programming language to a lower-level. The output of a compiler is called object program and the input to a compiler is called source program. Figure 3.1, gives a graphical presentation of the process [Sebesta, 2013].

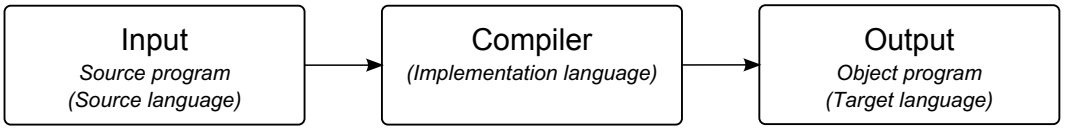


Figure 3.1. Simplified overview of the compilation process.

Compilers are able to catch some errors in the source program during the compilation process. It checks if the source program is a valid program in the language and it will report if it finds invalid code in the source program. A compiler is also able to optimize the source program, thereby creating faster code [Sebesta, 2013].

A compilation process can be really slow. This is a problem for large code bases that need to be compiled, if these are not used [Sebesta, 2013].

Interpreter

An interpreter is a program that interprets and executes programs written in some programming language. As opposed to a compiler, it does not translate the code to another language. Interpreters scan the source program and execute it one statement at a time. Figure 3.2, gives a graphical presentation of the process [Sebesta, 2013].

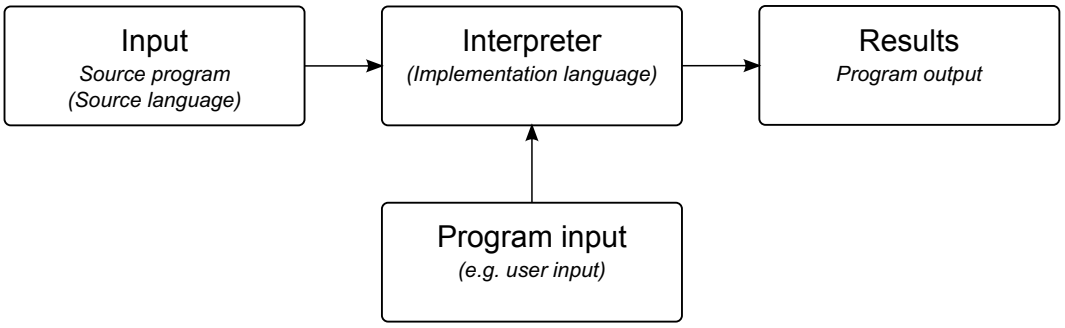


Figure 3.2. Simplified overview of the interpretation process.

Interpreters have to interpret a program every time a program is executed. The interpreter can only catch errors at runtime. A consequence of these is a slower execution time for interpreters, compared to a compiler. As an example, it is used when other operations makes the executions slow, like user inputs. Another example that could slow down the execution time could be floating point calculation or the network connection.

3.4.1 Key differences between compiler and interpreter

This section attempts to highlight some of the key differences between compilers and interpreters and the differences between the results they produce from source code. Table 3.2 show an overview of the differences between compilers and interpreters.

Characteristics	Compiler	Interpreter
Control	Program	Interpreter
Implementation	Harder	Easier
Runtime speed	Faster	Slower
Debugging	Compile Time and Run Time	Run Time
Memory usage	More	Less
Translation	Translates to target code	Translates to actions
Optimization	Yes	No

Table 3.2. Differences between a compiler and an interpreter [Fischer et al., 2009; c4learn, 2014].

One of the biggest differences at runtime between a compiled program running directly on the target machine hardware and an interpreted program running in a virtual environment is what is in control. For interpreted programs, execution control is actually in the interpreter and not in the running program, as opposed to a compiled program that runs directly on machine hardware.

Another difference is the memory usage of a running program. The compiler use more memory on execution than a interpreter because all the object code needs to be load intro memory. An interpreter only need one line of source code at a time in memory Teach-ict [2014].

Compiling an entire program leaves room for optimization of the entire program. A compiler sees the whole picture of the program and it is therefore possible to optimize the program. A compiler might for instance decide to unroll a loop, i.e. copy the instructions executed in the loop, whereas a interpreter will not because it does not know the context in which it is executing its statements Stefan Brunthaler [2012].

3.4.2 Language processor choice

Programs written in PIMPL can be set to run once or to run recurrently, as mentioned in section 3.1. It makes sense to make use of an interpreter if programs are set to be run once and are then discarded. It is not necessary to wait for a program to be compiled in order to run it, when utilizing an interpreter.

Interpreting programs is slower at runtime, but the total time could be shorter, if a program only should run a single time. It is faster to compile the program once, rather than interpreting the program every time it is executed. The slow interpretation does not mean so much for programs written in PIMPL, because the time-consuming part of a PIMPL program is the file operations it performs, rather than the compilation process.

If a PIMPL program is set to recurring, it runs in the background and uses the same code multiple times. For this, it can make sense to compile the program, resulting in faster execution. It would be practical if the language processor could look at the code, and then decide if it should compile or interpret the program. These are the characteristics of a hybrid language processor. But due to the time limitation and experience on the subject, this has not been chosen.

It is harder to implement a compiler rather than an interpreter, but the benefits of this result in debugging done at compile time, where some errors can be avoided. A compiler also has the ability to optimize the program code when translating to the target code.

The project group has chosen to make a compiler, as this performs the best for recurring programs, and can still run when programs is set to run once.

When compiling, it is necessary to find a target language. For this project, this has been chosen to be C#, because PIMPL need to make file operations in Windows, and C# works well with Windows APIs, as Microsoft has created this language. C# is also a language the project group is fairly experienced with.

Compiling to a high level language, makes some of the code generation easier, because its easier to express certain methods in these languages.

A more optimal solution would be to write to a more low level language such as Java bytecode or CIL, however the project group felt that getting acquainted with a new language would be too time consuming.

3.5 Preliminary choices summary

To outline all the choices made in this chapter, a brief summary follows.

The language evaluation criteria has been chosen and prioritized, ranging from very important to not important. The primary focus of these criteria is on the readability criteria, where learnability supports this. Other important criteria are implementability, writability and reliability.

The paradigm chosen is the imperative paradigm, where the declarative paradigm also were considered. Though problems with the declarative paradigm and the sequence in which the rules should run could become a problem. Therefore the paradigm choice is the imperative.

If has been chosen to develop a compiler, rather than an interpreter, as the compiler performs better if the programs are set to recurring. The language that the compiler translates to is C#.

These choices are used to design and implement the PIMPL language.

Design and syntax 4

PIMPLs design and syntax is describes in this chapter. In this context some of the semantics is described informally, where chapter 5 described the formal semantics.

The full Extended Backus-Naur Form(EBNF) context free grammar(CFG) of PIMPL is listed in appendix B. This chapter only presents small parts of the CFG as examples. The full list of lexical elements of PIMPL can be seen in appendix D.

Before PIMPL can be described, it is needed to describe some theory about the syntax, and the format of the grammar in PIMPL. After this the general structures of the language are described e.g. comments, the different types and the operators for these types. Later the focus is on the two main parts of the language, the Rule-part and Library-part. Some sections are moved to appendix E, as these are elementary for a number of programming languages.

4.1 Syntax theory

Syntax is about how a programming language is expressed. A programming language consist of a set of strings of characters from some alphabet. The syntax rules defined for the language defines which strings of characters, from the languages alphabet, are allowed in the language.

In a programming language, characters are referred to as lexemes, the lowest level syntactic units. Tokens consist of a group of lexemes or one lexeme. As an example, the token ID, is names of variables and subprograms [Sebesta, 2013]. Listing 4.1 shows a small syntax example from PIMPL.

```
index = 2 * count + 17;
```

Listing 4.1. Syntax example.

Listing 4.1 includes eight tokens. There are two ID's, "index" and "count". The =(Equal symbol), *(Multiply symbol), +(Plus symbol) and ;(Semicolon symbol) are all lexemes. Both of the numbers, "2" and "17", are also tokens.

The syntax is described with a list of tokens, and a CFG. The lexemes are described with regular expressions, and the CFG is in EBNF.

Format of EBNF

The CFG is a list of production rules, containing terminals and nonterminals. Here nonterminals is described with a production of terminals and nonterminals. Listing 4.2 shows an example of a production.

```
<Values> = [<Value>] { comma <Values> }
```

Listing 4.2. The example of a CFG production, in EBNF form.

The nonterminal **<Values>** on the left hand side, of the equal symbol, is described with the right hand side with the list of the nonterminal **<Value>**, the terminal **comma** and the nonterminal **<Values>**.

The EBNF is used with the syntax in table 4.1. The table includes syntax only allowed in BNF, and the additions in EBNF. An example of use can be seen in listing 4.2.

BNF Syntax		**EBNF Syntax**	
Definition	=	Optional	[...]
Alternation	|	Repetition	{ ... }
Nonterminal	< ... >	Repitition, at least once	{ ... }+
Comment	(* ... *)	Grouping	(...)
End of program	$		

Table 4.1. Table showing the BNF and the EBNF syntax.

4.2 Comment

PIMPL includes the possibility to include comments in the source code files. The project group has decided to use a # (number sign) to define the start and end of a comment. Because the group thought that the symbol would be easily recognizable, as it is not used in normal text. It is not possible to use the # symbol in a comment, as this will cause the comment to end. Everything in between these two symbols is seen as a part of the comment, and will not be considered further when processing the source code. Languages such as Perl, Ruby and Python also uses the # for comments. An example of a comment can be seen in listing 4.3.

```
# This is a comment for the language PIMPL #
```

Listing 4.3. An example of a comment in PIMPL.

4.3 Types and Values

A number of different types are available in the PIMPL language, and some values associated the types. Operators for the types will be described later in section 4.5.

The following types exist in PIMPL, and will be described in detail in the following sections:

- String
- Int (Integer)
- Bool (Boolean)
- Path
- Date
- Time
- RegEx (Regular Expression)

The String, Int and Bool types are included in PIMPL as a way to work with these elementary types. Path, Date and Time are specific to PIMPL, and are required for file manipulations along with the other types. RegEx can be used if the user wants to manipulate the other types.

4.3.1 Variables

Variables can be declared and used in the Library-part, but only in subprograms like Selectors and Calculators and not globally. The syntax to declare variables is shown in listing 4.4, where T is a type, ID is the name and Value is a value of the given type.

```
T ID = Value;
T ID;

Int TheAnswer = 42;
```

Listing 4.4. The template syntax of a variable declaration and a concrete example.

It is not required to assign the variables a value. In case the variable is not assigned a value, it will be set to a default value. The "undefined" value is used when no logical default value exist. The default values of each type can be seen in table 4.2.

Type	Default value	Type	Default value
String	"" (Empty string)	Time	00:00
Int	0	Path	undefined
Bool	False	RegEx	undefined
Date	01.01.1970		

Table 4.2. Default values for each of the available types.

4.3.2 Elementary types

In PIMPL there are three elementary types, String, Int and Bool. These three types have been named from their values. The names have been shortened e.g. *Integer* is called *Int*. These three types are similar to those of other programming languages, and is therefore only described in short here. All the elementary types are described in detail in appendix E.

A String is a ordered collection of characters in a static order. In PIMPL, all the characters allowed in file names on Windows are allowed to be used as part of a String. An Int is an Integer which is able to contain numbers from the range -2^{31} to $2^{31}-1$, as this in the range in the targeted language. In PIMPL an Int is used for counting and conditions. A Bool is able to either evaluate into "True" or "False", but these can also be stated respectively as "Yes" and "No" in PIMPL.

4.3.3 PIMPL specific types

Besides the elementary types, there is some types that is special for PIMPL. They are described in the following sections.

Path

A Path are often used in PIMPL, as the purpose of the programming language is to organize the file system. It is possible to write a Path in different types. Although, in some places of the program, it is required to provide a specific Path type. There are four different path types, and these are the following:

- Absolute path
- Relative path
- Absolute path with file specification
- Relative path with file specification

These different path types all go under the common type Path. An example of every path types is shown in listing 4.5.

As seen, absolute paths shows the entire path from the computer volumes root folder, to the wanted folder, whereas a relative path is related to an absolute path set earlier. When specifying a WorkFolder to work in, this path must be an absolute path.

Any Path is surrounded by quotation marks("). An absolute path starts with the drive symbol (e.g. C), followed by a colon(:) and a backslash(\). The relative paths later in a program have the defined WorkFolder Path as prefix, denoted by a star(*) symbol, followed by a backslash(\). This is followed by a folder name which again is followed by a backslash and name, for each subfolder. The two path types with file specification, does not end at a folder, but ends at a specific file, as seen in listing 4.5. All the different path types can be used in the Exclude statement and Do-operation, as shown in section 4.6.

```
Absolute path:            "C:\Users\Inigo Montoya\Pictures\"
Relative path:            "*\Pictures\Stuff\"
Absolute path with file:  "C:\Users\Inigo Montoya\Pictures\Cat Wallpaper.png"
Relative path with file:  "*\Pictures\Stuff\Something.jpg"
```

Listing 4.5. Examples of all the path types.

Date and Time

When organizing files, users might want to sort pictures by a given date. The form of these can be in Date and in Time, which can be perceived alike, and the differences will

be described in the following sections.

Date
A Date is of the form dd.mm.yyyy, where the order is fixed, and dots(.) can be replaced by dash(-) or forward slash(/). Rules apply so that the user is unable to declare month as a negative number and not higher than 12. Similar rules apply to day and year, with appropriate boundaries.

```
Select FilesBetweenDates(02.06.2013, 16.06.2013);

...

Selector FilesBetweenDates(Date start, Date end)
{
    If (ThisFile('DateCreated') > start AND ThisFile('DateCreated') < end)
        Selected;
    Else
        Deselected;
}
```

Listing 4.6. PIMPL code: The use of Date in a Select and the associated Selector.

As seen in listing 4.6, elements of the type Date is being used to select files between two specified dates. If a file is created within these two dates, it will be selected. The Selector declarations are further described in section 4.7.3.

Time
The use of Time is similar to the use of Date. The only difference is how the input is formatted. An example of Time can be seen in listing 4.7. All the pictures captured between the time 14:00 and 18:00 are selected, regardless of the date captured.

```
Select FindByTimeSelected(14:00, 18:00);

...

Selector FindByTimeSelected(Time start, Time end)
{
    If (ThisFile('DateCreated') > start AND ThisFile('DateCreated') < end)
        Selected;
    Else
        Deselected;
}
```

Listing 4.7. The use of Time in a Select and the associated Selector.

RegEx

Regular Expressions in PIMPL are targeted the more experienced users, this feature makes it possible to see if e.g a string follows a certain pattern.

Regular Expressions in the PIMPL language use the same syntax as in other programming languages such as Java, and C# [Oracle, 2014; Microsoft, 2014c]. Regular expressions in PIMPL are designed to pattern match exactly like they do in C#, this should make it easier for experienced users to use Regular expressions in PIMPL. A Regular Expression starts

with the quotation mark symbol(") followed by forward slash (/)and end with the same symbols, in reversed order. Listing 4.8 shows an example with a regular expression.

```
If (ThisFile('FileName') == s + "/[a-zA-Z]*/")
    Selected;
```

Listing 4.8. An example of a RegEx.

4.4 ThisFile

ThisFile is conceptually a reference to the current file the program is working on in the current file. In the Rule-part ThisFile only allowed in Do declarations and in the Library-part ThisFile is allowed in Selectors and a Calculators ForeachFile loops. The keyword ThisFile can be used in three different language constructs: ThisFile(MetadataSpecifier), ThisFile.HashValue and ThisFile.IsSelected. In the Rule-part only the MetadataSpecifier construction is allowed. For an example see listing 4.9.

```
If (ThisFile('Photo+CameraManufacturer') ==  "Sony") ..
```

Listing 4.9. An example of ThisFile.

MetadataSpecifier

Metadata can be identified with a MetadataSpecifier. A metadata is a value of these types: String, Date, Time or Int. The MetadataSpecifier has been added, as the restrictions that String has, should not apply to these. A MetadataSpecifier is used when metadata from files is needed in the users programs.

MetadataSpecifier is surrounded by single apostrophes(') at the beginning and at the end of a string of characters. The string can contain all symbol except apostrophe. When the users access metadata, they must use an additional keyword, to state what categori of metadata that needs to be accessed, and a plus symbol(+) that should specify exactly what metadata the user want. MetadataSpecifiers can only be used in the context of ThisFile(), where it should be placed in the parenthesis. The mappings between the MetadataSpecifiers and the types used for the values in PIMPL, can be seen in appendix I. Listing 4.10 shows an example of the use of a MetadataSpecifier.

```
String s = ThisFile('FileName');
# Finds the metadata with the identifier Name for the current file #

Date d = ThisFile('Photo+DateTaken');
# The 'Photo+DateTaken' takes the DataTaken from the Photo category of
    metadata #
```

Listing 4.10. The use MetadataSpecifier.

HashValue

ThisFile.HashValue returns a hash value for the current file. We chose the hashing algorithm RIPEMD-160 [Antoon Bosselaers, 2012]. The HashValue can be used to compare two files. The content of two files can be considered the same if the hash values of the two files are the same. There exists a small possibility that two files with the same hash value are not identical, this is called a hash collision, for RIPEMD-160 a hash collision has still not been found [Antoon Bosselaers, 2012].

IsSelected

IsSelected is an option for checking if the current file is already selected. This option is only allowed in the Selector, because it is the Selector that selects files. The value of IsSelected is a Bool value. True means the file is selected and false means the file is deselected. Files are deselected as default.

4.5 Operators

There are a number of operators in PIMPL, which can be used with different types in the language. Some of these operators work with most types, like the equal and not-equal operators, and some operators are specified to certain types. In the following subsections, the different operators are described in detail. The description includes the syntax of the operators, how the operator are used, why the syntax look like it does, and how it informally works with a given type.

In the sections, when 'a' and 'b' are used as variables that can represent any of the types in PIMPL.

All the operators, for the different types, are detailed in this chapter. But first a presentation of the precedence of the operators in PIMPL is shown. Following this is a section about coercion, that describes the possibilities of implicit casting, associated with the concatenation operator.

4.5.1 Precedence

In PIMPL, the operator precedence is similar as in the C language, but PIMPL includes less operators than in the C language. The precedence in C is based on the precedence in mathematics. The choice of this operator precedence, is to make it familiar for the target group. In table 4.3, the operators precedence is shown. The unary operators are calculated from right to left, while the rest are calculated from left to right.

Precedence	Operators				Notes
1	( )				
2	NOT	-			Unary
3	*	/	%		
4	-	+			
5	<	>	<=	>=	
6	==	!=			
7	AND				
8	OR				
9	,				

Table 4.3. The precedence of the operators in PIMPL.

4.5.2 Coercion

Coercion is a language ability to implicit cast different data types into other data types. The relationship between data types in PIMPL is shown in listing 4.11. The plus(+) operator has different meaning depending on the types it is used with. When the plus operator is used as concatenation, coercion can be used.

```
Date
Int    < String < Path < RegEx
Time
```

Listing 4.11. Relationship of types in PIMPL.

A String can be concatenated with a Path, this results in a new Path, as a Path includes more symbols than the allowed symbols in a String. The order of the concatenation is important, which can be seen in listing 4.12.

```
Path A = "C:\Users\InigoMontoya\Pictures\" + "Starlight"; # Legal
    concatenation #
Path B = "Starlight" + "C:\Users\InigoMontoya\Pictures\"; # Illegal
    concatenation #
```

Listing 4.12. Example of legal and illegal casting of String and Path.

Coercion can change narrow types to wider types if necessary, as shown in listing 4.11. This could be when two different type's values is concatenated, assigned to another type or compared. As an example see listing 4.13.

```
String s = 30 + "Example text" + 4 + " ";
If (s == s + "/[0-9]*/") ...
```

Listing 4.13. Example of coercion with use of concatenation.

In PIMPL, coercion is used to combine values of different types.

4.5.3 Elementary type operators

As mentioned in section 4.3.2, PIMPL includes three elementary types, Int, String and Boolean. These three elementary types can be used with a number of operators. A full description of those operators are shown in appendix E.

4.5.4 PIMPL specific type operators

With the PIMPL specific types a number of operators can be used. This section contains a description of the behavior of the different operators.

Operators on Path values

On the Path types concatenation(+) can be used.

1. a + b

However concatenation of two paths is not possible, because this will not result in a valid path. The concatenation can be used with a string/int and a path, and thereby give a new path, see example in listing 4.14.

```
String s = "Documents";
Path a = "C:\Users\Username\" + s + 4 + "\";

# Result: C:\Users\Username\Documents4\ #
```

Listing 4.14. Concatenation with paths.

Operators on Date and Time values

The operators on Date and Time values are comparison operators, as seen in the list below. The operators are the same as some of the integers operators. These operators are used to compare different Date and Time variables. There are no arithmetic operators, because this does not make any sense, as this easily can result in dates far in the future. A comparison has to be with values of the same type.

1. a < b
2. a > b
3. a <= b
4. a >= b
5. a == b
6. a != b

RegEx operators

There are three operators that works with RegEx variables. For the comparison operators(== and !=), only one of the operands can by These operators are listed below.

1. a + b
2. a == b
3. a != b

It is possible to concatenate (+) a RegEx with all the types in PIMPL. It is also possibly to compare a Regular Expression with one of the other types, to check if it matches the RegEx. This is shown in listing 4.11.

4.6 Rule-part

The Rule-part contains the Options and Rules, where all the options have to be defined in the beginning of the program. The Options are defined at the beginning of a Rule-file in order to set the frames for the Rules. In this section both the Options and the Rules are described in details.

4.6.1 Options

This section describes some of the Options, both the mandatory and optional, for a program. These options are set in the Rule-part of a program. The options will be described in their associated section in this section. Listing 4.15 shows the **<Option>** production in the EBNF grammar of PIMPL.

```
<Options> = <IncludeLibraryStmt> <WorkFolderDef> <SetRepeatOptDef>
    <WorkInSubOptDef> [<Exclude>] [<RunRules>] [<FinallyOnChange>] [<SetDcls>]
```

Listing 4.15. The EBNF for the options.

Following this is a table 4.4, which gives a simple overview of which options are mandatory and the order in which they must appear.

Option	Mandatory
Use	Yes
WorkFolder	Yes
RunOption	Yes
WorkInSubFolders	Yes
Exclude	No
RunRules	No
FinallyOnChange	No
Set	No

Table 4.4. Possible options in Rule-part and their order.

Use

The first option is to define the use of Libraries. Libraries is described in details in section 4.7. The Use option makes it possible for the users to decide what Libraries they want to use. Users are able to define their own Libraries, which means that there should be a way to use these in their programs. The syntax is the keyword “Use” followed by a list of Library file names, that are separated by commas (,) and ended with semicolon (;). An example can be seen in listing 4.16.

```
Use "MyOwnAboutImages.lib", "MyOwnAboutDownloads.lib";
```

Listing 4.16. Example of Use Library.

It is not possible to select files without the use of a Library, because the Select declaration must make use of a Selector, which are declared in the Library.

WorkFolder

WorkFolder defines a Path in which a program should operate at, which works as a root directory for the program. From the WorkFolder declaration follows a list of files that the Rules works on. Rules can then move or delete files from this list, which means that the list is modified after a Rule has been run. A definition for WorkFolder is mandatory to define, while using the language. It is required to state a WorkFolder, in order to avoid file operations in undesired folders. If no WorkFolder is defined, then the program does not know where to operate, and no file operations can be done. To see how a WorkFolder is declared in the code, an example have been provided in listing 4.17. It is not possible to have two WorkFolder definitions at the same time. An absolute path must be used to define a WorkFolder.

```
WorkFolder "C:\Users\Username\Pictures\";
```

Listing 4.17. Example showing WorkFolder with a Path.

RunOption

It is mandatory to define the RunOption. RunOption is the option that states how a program should run. PIMPL programs can, with the RunOption, be set to run "Once" or to run "Recurring". There does not exist a default RunOption. If no RunOption is defined, the program will not compile. The RunOption needs to be defined in order to increase the readability and reliability of programs. "Once" means that the program should only run once and then terminate. "Recurring" means that the program reacts on changes in the WorkFolder.

In listing 4.18 an example of the RunOption is shown. In the comment is the second option, which the RunOption can use.

```
RunOption Once;  # Recurring #
```

Listing 4.18. Shows RunOption set to Once.

WorkInSubFolders

WorkInSubFolders exists in order to make the programmer able to define if the program should include subfolders of the specified WorkFolder path. All the files from the WorkFolder's subfolders are added to the list of files, that is being worked on, if the WorkInSubFolders option is set to "True". It is mandatory to define if the programs should work inside subfolders. This construction will give the user better control over which files the program is working on, and this increases readability and reliability.

The code example in listing 4.19, shows an example of the WorkInSubFolders option.

```
WorkInSubFolders True;
```

Listing 4.19. Shows an example of WorkInSubFolders.

Exclude

It is optional to define folders or files that should be excluded from the list of files being worked on. If files or paths are entered into the exclude option, then these files or paths are removed from the list of files, that is being worked on. It is possible to use all the Path types to define what should be excluded. This options is optional because the user does not always want to exclude some paths.

Listing 4.20 presents how a single path is defined, and how multiple paths/files are excluded.

```
Exclude "C:\Users\Username\Pictures\Stuff\";

Exclude "C:\Users\Username\Pictures\Stuff\", "*\Docs\Stuff.txt";
```

Listing 4.20. Excluding paths by using two different Path types.

RunRules

RunRules gives the opportunity to define the sequence of which the rules is run, if the Rules should be run in a different order than the rules are written in. It also gives the opportunity to choose only to run some of the rules. It is optional to define RunRules, but it gives some extra possibilities in the way to order the rules in a program. The RunRules increases the readability and reliability of programs. This possibility makes it simple to reorder rules by just switching declared rule names instead of entire rule code blocks. The rules the user wants to be run should be listed by ID after the keyword RunRules, and separated by comma (,) and ended with semicolon (;). Only the rules in the list runs, and in the order they are listed. If no definition of RunRules exist, the Rules run in the order these Rules are declared.

```
RunRules MyRule1, SortFlowers, CleanUpEverything;
```

Listing 4.21. Shows RunRules with specified rules.

FinallyOnChange

FinallyOnChange gives the option to be notified when one or more Rules have been enforced. It is optional to define this option. The FinallyOnChange code runs once all Rules have been completed if one or more file operations have been executed.

Listing 4.22 shows two code examples, one for the Beep() and one for the Message() construction. These are some small notifications that can be heard/shown when the program is done executing. The Beep() will play a little beep noise and the Message() will print out a message written inside the parenthesis. The Beep() option is used, when the

user want to be informed in a relatively discrete way, while the Message() can say more about what is happing. If a user has more than one program running on different work folders, a Message() can e.g. tell which program finished.

```
FinallyOnChange Beep();

FinallyOnChange Message("One or more files have been changed in my document
    folder.");
```

Listing 4.22. Shows FinallyOnChange with Beep() and Message().

Set declaration

Set is a collection of elements (values) of the type T, and can only contain values of this specified type. T can be any type available in PIMPL. The order of values in a set does not matter. The Sets can only be used as the last parameter in a Selector call. Sets are used to see if for example a file type is a picture file type, see listing 4.23, that also shows the syntax. As seen in the listing, using a Set in a Select, work the same way as calling the same Selector multiple times separated with OR between the different elements in the set. It increases writability, because the user can make more comparison at the same time. The use of Set makes more sense if there are more rules that works on for example the same file types, because it ensures that the user always use exactly the same file types when it is wanted.

```
Set<T> ID(value_1, value_2, ..., value_n);
...
Select Selector1(ID);

# Equivalent with the following #

Select Selector1(value_1) OR Selector1(value_2) OR ... OR Selector1(value_n);
```

Listing 4.23. A presentation showing the syntax of Sets.

Multiple Sets can be defined at this point in the code, and it is optional for the user to define any.

4.6.2 Rules

The Rules contains the two parts Select and Do. It is in the Rules that the users can define how to select files and what operations should be done to these files. There should be defined at least one rule in a program and it is possible to define multiple rules if needed. If a rule is not defined in a program, the program cannot be compiled.

Listing 4.24 shows a complete Rule example containing a Select and a Do. The Select and Do declarations are described in the following sections.

```
Rule MyRule1     # MyRule1, is the ID of the Rule #
{
    Select DublicatesByName();
    Do Delete();
}
```

Listing 4.24. An example of a Rule.

The rules will be run in the order they are defined, unless otherwise is specified with the RunRules option. More precisely the topmost rule will be run first, then the second topmost rule and so forth. This could possible mean a lot, as an example: when one rule say to move all .png files, and another rule says to delete everything, then the user probably want to move their .png files first, and afterwards delete the rest.

Select

The Select declaration is the first part of a Rule. The Select declaration is the way to define which files to make file operations on. The Select statement consists of one or more Selectors defined in the Library-part.

Between the Selectors the user can use the OR and AND logical operators. Every Selector will conceptually return a list of selected files which are used in the second part of a Rule, the Do declaration. If AND is used between Selectors the list from the Selectors will be compared and only the files that exist in both of these list will be selected. If OR is used between Selectors, all files in both lists are selected. It is also possibly to use NOT in front of a Selector, and the result are the inverse list of the Selector.

When declaring a Select, the user must write the keyword "Select" followed by a Selector and the actual parameters for these, if any, and ended with a semicolon(;).

```
Select MostFrequentFileType() AND NamePrefix("Blomst");

Select FilesBetweenDates(02.06.2013, 16.06.2013) OR NamePrefix("Blomst");

Select All();
```

Listing 4.25. Example of three Select declarations.

Examples of a Select is shown in listing 4.25. The example, two different Selects are shown. The first one selects all files selected by the MostFrequentFileType and NamePrefix Selectors. These are both needed to be specified in a Library. The Selectors used will be further described in section 4.7.3. The MostFrequentFileType selects all the files of a certain type, like .jpg or .png, and the second Selector NamePrefix, selects all files which begins with the name "Blomst". These two list are compared, and if the file type is the most frequent, as stated by the first Selector, and if the file name begins with "Blomst", then the files are chosen, and the file operations specified by the Do declaration will be executed on these file. The second example uses the Selector FilesBetweenDate or the Selector NamePrefix, meaning that the two lists returned from the Selectors are combined and all the files from both is Selected. The last example selects All files in the WorkFolder.

Do

As described in section 4.6, a rule consist of a Select and a Do. The Do describes what should be done with the files selected in the Select declarations. The Do is limited to a fixed set of available file operations, these operations were described in section 2.2.1, where an analysis found that these are available relevant file operations in a Windows operating system. These file operations and their types are:

- **Move** - The move will take any Path constant, absolute or relative, and the Path will always be written using quotation marks(").
- **Rename** - The rename can take user made strings, written inside quotation marks, these may be concatenated with other strings, or metadata by using the ThisFile(), using the plus(+) operator.
- **Delete** - The delete does not take any parameters.
- **Copy** - Like move, the copy will take any Path constant, absolute or relative.

The Do is started by typing the word "Do" followed by the available file operations with paranthesis containing the parameters followed by a semicolon(;).

And the example of a Do is shown in listing 4.26. Here, the first example shows a Delete() operation, which deletes all the files selected by the Select operation. The next example consist of two different file operations, namely Move and Rename. Here, the selected files will first be moved into a different folder (If this folder does not exist, then it is created). Then the files are renamed into "SomeThings" followed by the files' creation date. The list of files in the WorkFolder is updated when every Rule has concluded.

```
Do Delete();

Do Move("C:\Users\InigoMontoya\Things\"), Rename("SomeThings" +
    ThisFile('DateCreated'));
```

Listing 4.26. Some examples of Do operations

4.7 Library-part

This section describes the Library-part of PIMPL. A Library file is specified by writing "Library:" in the start of the file. The Library-part consist of subprograms namely Selectors and Calculators, which allows the users to specify files. It is not allowed to have nested subprograms, and the subprograms must be declared before they can be used. As mentioned in section 3.1.1, the Library-part is focused on the experienced users. The Library-part contains statements similar to languages like C and Java, which are typed at a lower abstraction level compared to the statements in the Rule-part.

The general building structures, namely ForeachFile loops, If-Else statements and Persistent variables are described in this section. After this, the Selectors and Calculators are described in more detail. The building structures are mentioned first, as some of these can be used in both Selectors and Calculators.

4.7.1 General building structures

To control the flow of a program different kinds of control flow statements used in the Library-part. These statements are ForeachFile and If-Else statements which both are described in the following sections. It is also relevant to describe Persistent variables in this context, which is described after the statements.

ForeachFile

ForeachFile is used when the user wants to look through all the files in the WorkFolder. When WorkInSubFolders, as described in section 4.6.1, is set to true, ForeachFile will see all files as one long list, which is independent of which folders these files are placed.

With ForeachFile we took inspiration from the C# foreach statement, but instead of doing something with each element in a Enumerable collection, ForeachFile executes statements for each file in the WorkFolder. Where the WorkFolder can contain additions from the WorkInSubFolders option, or exclusions from the Exclude option.

ForeachFile works by having the ForeachFile statement followed by a statement the user want to perform on all the files in the folder. Listing 4.27 shows an example of a ForeachFile statement, followed by If-Else statements. The If-Else statements are explained in section 4.7.1. If more than one statement is desired inside the ForeachFile statement, then it is necessary to use a block statement to place the statements in. This is done by placing braces ({ and }) after the ForeachFile statement, that encapsulates the other statements.

```
ForeachFile
{
    If (ThisFile('FileExtension') == ".png")
        NumberOfPng = NumberOfPng + 1;
    Else
    {
        If (ThisFile('FileExtension') == ".jpeg")
            NumberOfJpeg = NumberOfJpeg + 1;
    }
}
```

Listing 4.27. Example showing the ForeachFile followed by an If-Else statement.

If-Else

The If-Else statement are borrowed from the If-Else statement from languages such as C, C# and Java. An If-Else statement starts with an If-statement with parenthesis around the condition, followed by a statement, where a statement can be a block with statements.

Listing 4.27 shows an example of an If-Else statement being used in PIMPL, by comparing the number of .png and .jpeg files.

The PIMPL grammar contains a problem, the dangling else problem. This program is connected to the If-Else statement. The dangling else problem is when an optional else clause can result in nested conditionals to be ambiguous, which results in different parse

trees. This is solved by always connecting an else to the nearest if statement that does not contain an else.

Persistent variable

A Persistent variable is a fixed variable, which is set at the first run of a Selector. The listing in 4.28, shows a Persistent String fileType, which is set to the Result of the Calculator function MostFrequentFileTypeCalc. This Calculator will therefore only be called once, despite the Selector running on each file in the WorkFolder. Calculators are described more in section 4.7.4. If a Persistent variable is declared in a Foreach statement, then this value is assigned one time, while all other statements in a Foreach is run n^2 times. The Persistent variable is reset every time a rule executes.

```
Selector MostFrequentFileType()
{
    Persistent String fileType = MostFrequentFileTypeCalc();

    If (ThisFile('FileExtension') == Undefined OR ThisFile('FileExtension')
        == fileType)
        Selected;
    Else
        Deselected;
}
```

Listing 4.28. An example showing a Selector that uses an If-Else and a Persistent variable.

4.7.2 Subprograms

The Library-part contains subprograms, namely Selectors and Calculators. This section describes the formal- and actual parameters that is similar for both subprogram. After this, the Selectors and Calculators are described in depth. PIMPL supports parse by value parameters. The scope rules in PIMPL is static.

Formal parameters

When creating a subprogram in PIMPL, parameters can be specified to give the subprogram some parameters to work according to. These parameters are written after the function name in parenthesis. The code of listing 4.29 shows how formal parameters are written, here the two parameters are of type Date, and the two variable names in the Selector is called “start” and “end”. These can then be used in the body of the subprogram between the two braces.

```
Selector FilesBetweenDates(Date start, Date end)
{
    ...
}
```

Listing 4.29. Formal parameters of the Selector FilesBetweenDates.

Actual parameters

When a subprogram should be used, the users have to make a call to it. A call include a list of actual parameters if the subprogram definition requires it. The actual parameters show which value the subprogram should work with. The example in listing 4.30, shows two values, which has been sent into the subprogram namely "02.06.2013" and "16.06.2013". These two act as the "start" and "end" variables in the program, which was described in the previous section(4.7.2). The variable types in the actual parameters must match those in the formal parameters, and also the order of their appearance must match.

```
Select FilesBetweenDates(02.06.2013, 16.06.2013);
```

Listing 4.30. Actual parameters of the Selector FilesBetweenDates.

4.7.3 Selector

The Selectors are subprograms used by a Select declaration to choose exactly which files the user wants to make file operations on. Listing 4.28 shows an example of a Selector which selects files that are exactly the specified file type, or files where the type is not defined.

The following sections describe the connection between the Selects in the Rule-part and the Selectors in the Library-part.

Connection with Select

In order to select the files that the user wants to work on, a Selector must be used. This is done by using the Select in the Rule-part, that needs to use a Selector in the Library-part. Listing 4.31, shows how the Select chooses a Selector.

```
Select DublicatesByName();
```

Listing 4.31. The Select in the Rule-part which calls the Selector "DublicatesByName".

Selected/Deselected

Selected and Deselected are two statements connected with the current file (ThisFile) used in Selectors. The Selected and Deselected statements returns a value from Selectors, that describes whether or not a file is accepted and hereby operations can be done on these file. Listing 4.32, shows a Selector FilesBetweenDates, that checks whether or not a file's creation date is within two specified dates. If the file's creation date is between these dates then it is selected and if not it is deselected.

```
Selector FilesBetweenDates(Date start, Date end)
{
    If (ThisFile('DateCreated') > start AND ThisFile('DateCreated') < end)
        Selected;
    Else
        Deselected;
}
```

Listing 4.32. Example of Selected and Deselected.

The use of Calculators

Calculators can be used if the users wants to calculate some values to base their file selection on. These can be used if the users wants to calculate some values first, that may influence the file selection. This could e.g. be when the user wants to organize after which file type the user has the most of. A Calculator can then provide the information needed.

4.7.4 Calculator

A Calculator is a subprogram used in the Selectors in the Library-part. The syntax can be seen in listen 4.33, where T is any type in the language and ID is the name of the Calculator.

```
Calculator<T> ID()
{
   Statements
}
```

Listing 4.33. The abstract syntax of a Calculator.

Calculators are used to calculate some value which is to be used in a Selector, e.g. if it should choose .png or .jpeg files. The example in listing 4.34 shows a Calculator, that choses what filetype should be used as a organization basis. This is used to define what filetype is used in listing 4.28. The Calculator will return a result, like functions in C or C#. Much like these, Calculators will return a result, which will be used in Selectors. To return a result, the user should assign it to Result, which is an implicitly declared variable. The type of the implicit Result variable is the type denoted by the T in the declaration of the calculator.

```
Calculator<String> MostFrequentFileTypeCalc()
{
    Int NumberOfPng = 0;
    Int NumberOfJpeg = 0;

    ForeachFile
    {
        If (ThisFile('FileExtension') == ".png")
            NumberOfPng = NumberOfPng + 1;
        Else If (ThisFile('FileExtension') == ".jpeg")
            NumberOfJpeg = NumberOfJpeg + 1;
    }

    If (NumberOfPng > NumberOfJpeg)
        Result = ".png";
    Else
        Result = ".jpeg";
}
```

Listing 4.34. A Calculator.

Formal semantics 5

This chapter includes the most essential parts of the formal semantics of PIMPL. It is chosen to define formal semantics of PIMPL to avoid ambiguities in the semantics. A type system is included in the formal semantics, in order to make it possible to describe the different type rules of PIMPL.

There are several different ways to describe semantics formally, but the project group has chosen Structural Operational Semantics (SOS), as the group has attended lectures about this topic.

In the project the formal semantics is limited to only a part of PIMPL, because of a relative short project duration. It is chosen to focus on the semantics of the core parts of PIMPL, i.e. the selection and operations on files.

This chapter begins with a short section about SOS and type systems, followed by a choice between big-step SOS and small-step SOS. Afterwards the SOS and type system of PIMPL are described, including the transitions that the group finds most interesting, while the remaining rules defined can be seen in appendix G.

5.1 Theory

This sections describes the theory of SOS and type systems in short. Details are presented in the sections that uses them.

5.1.1 Structural operational semantics theory

SOS is based on the syntax of the language, and therefore the syntax needs to be mentioned in relation to the SOS. In semantics all details of the syntax are not relevant, as syntax analysis is of no interest here. An abstract syntax can therefore be made, where details as for example precedence is not included. This syntax description includes syntactic categories and formations rules.

The fundamental syntactic elements of PIMPL is represented by syntactic categories. An arbitrary element of a syntactic category is represented by metavariables. Elements of the syntactic categories are defined by a set of formation rules.

To define a SOS, it is also relevant to define environments, so it is possible to describe states in programs. In this context it should be defined how to specify a state and how to update a state. Auxiliary functions can also be necessary to define, to help writing the semantic rules [Hüttel, 2010].

An important part when defining a SOS is the transition systems. Transition systems define the meaning of the language, for example what statements and expression should do. These systems should define transition rules for all constructions in the language, which could be how to go from one state to another in the language. Transition rules all uses the template in figure 5.1, but premises and side conditions does not always need to be a part of a rule [Hüttel, 2010].

$$\frac{\textit{Premises}}{\textit{Conclusion}} \qquad \textit{Side conditions}^{1}$$

Table 5.1. Template of transitions rules.

5.1.2 Type systems theory

Type systems are a lot like SOS in the structure. Type systems are also dependent on the syntax. The core of a type system is the type rules for the different constructions in the language. Type systems also uses the abstract syntax from the SOS, where it is important that the types are included, in e.g. variable declarations. Besides this, extra syntactic categories for the types and some formation rules for these, are required [Hüttel, 2010].

Like with the SOS, an environment to describe the different states' types and some auxiliary functions, are required.

The type rules use the same template as transition rules in the SOS, as seen in table 5.1.

5.2 Big-step or small-step

There are two different ways to define a SOS, big-step SOS and small-step SOS. Big-step SOS describes an entire execution of a statement or expression, and small-step SOS describes parts of the execution of a statement or expression. Big-step SOS can be simpler than small-step SOS. A small-step SOS should be used if e.g. parallelism and infinite loops needs to be described. PIMPL only include one construction that cannot be described with a big-step SOS. This construction requires a small-step SOS in order to describe how the program never terminates. This construction is the "RunOption Recurring". Despite this construction, the project group has chosen to define a big-step SOS [Hüttel, 2010].

5.3 PIMPL's structural operational semantics

This section includes the central parts of the SOS of PIMPL. First the abstract syntax is described, where the syntactic categories and formation rules are shown and explained. Afterwards the relevant environments are defined, this includes the environments for ID, variables and subprograms, and the environment for files. At last the transition system is defined and a part of the rules are presented.

[1] In this report, some side conditions are placed underneath the conclusion.

5.3.1 Abstract syntax

These sections describe the abstract syntax of PIMPL, which includes the syntactic categories, their actual values, and semantic functions to map between them. Afterwards the formation rules for statements and expressions are presented.

Syntactic categories

There are different syntactic categories in the language and these can be seen in table 5.2. We assume that Ints are numerals of both positive and negative integer numbers. String consist of letters from the Latin alphabet, though with some exceptions, namely those that cannot exist in the file system in Windows. The actual values of these syntactic categories can be seen in listing 5.1, where they are described using regular expressions. The semantics of the syntactic categories is given by semantic functions (also shown in table 5.2), that maps e.g. n to integers or s to strings of characters. The actual values of the syntactic categories is also presented in the table.

Syntactic categories	Semantic functions	Actual value
$n \in Int$	$\mathcal{N} : Int \longrightarrow \mathbb{Z}$	$nv \in \mathbb{Z}$
$s \in String$	$\mathcal{S} : String \longrightarrow \mathbb{S}$	$sv \in \mathbb{S}$
$b \in Bool$	$\mathcal{B} : Bool \longrightarrow \mathbb{B}$	$bv \in \mathbb{B}$
$p \in Path$	$\mathcal{P} : Path \longrightarrow \mathbb{P}$	$pv \in \mathbb{P} \cup \{undefined\}$
$t \in Time$	$\mathcal{T} : Time \longrightarrow \mathbb{T}$	$tv \in \mathbb{T}$
$d \in Date$	$\mathcal{D} : Date \longrightarrow \mathbb{D}$	$dv \in \mathbb{D}$
$re \in RegEx$	$\mathcal{RE} : RegEx \longrightarrow \mathbb{RE}$	$rev \in \mathbb{RE} \cup \{undefined\}$
$ms \in MetaSpec$	$\mathcal{MS} : MetaSpec \longrightarrow \mathbb{MS}$	$msv \in \mathbb{MS}$
$id \in ID$		
$e \in Exp$		
$S \in Stmts$		

Groups	Semantic functions	
$dit \in DIT = Date \cup Int \cup Time$	$\mathcal{DIT} : DIT \longrightarrow \mathbb{DIT}$	$ditv \in \mathbb{DIT}$
$all \in ALL = Int \cup String \cup Bool \cup Path \cup Time \cup Date \cup RegEx$	$\mathcal{ALL} : ALL \longrightarrow \mathbb{ALL}$	$allv \in \mathbb{ALL} \cup \{undefined\}$

Table 5.2. The syntactic categories and the associated functions.

Table 5.2 shows the syntactic categories of PIMPL and mapping functions to actual values of the syntactic categories. The last row of table 5.2 shows a grouping of three syntactic categories which allows these three syntactic categories to be referenced under the same name. These groupings are defined in order to allow transition rules where elements of a group of different syntactic categories is evaluated to an actual values.

```
ℤ = /[-]([0-9])+/

𝕊 = /"[^\\/\*\?\"\|\:\<\>]*"/

𝔹 = /(True|Yes|False|No)/

ℙ = /"([a-zA-Z]:|\*)\\([^\/\*\?\"\|\:\<\>]+)(\\|(\.[^\/\*\?\"\|\:\<\>]*))"/

𝕋 = /([01]?[0-9]|2[0-3]):[0-5][0-9]/

𝔻 = /[0-9]{2}(\/|.|-)[0-9]{2}(\/|.|-)[0-9]{4}|/

ℝ𝔼 = /"/(~[/])+/"/
```

Listing 5.1. Regular expressions for the actual values.

We assume that there for every syntactic category exists semantic functions (See table 5.2), that maps each element of a syntactic category to the actual value of that element.

Members of syntactic categories are called metavariable and these are underlined, so that they may be distinguished from their corresponding actual values.

For instance, for the syntactic category Int, we assume that there exist a function $\mathcal{N} : Int \longrightarrow \mathbb{Z}$, that for each Int metavariable returns the actual value of that particular Int metavariable. An example of integer metavariable mapping: $\mathcal{N}[\![\underline{42}]\!] = 42$ and for String metavariable mapping $\mathcal{S}[\![\underline{Inigo\ Montoya}]\!] = Inigo\ Montoya$.

Formation rules

This section shows the formation rules of the PIMPL language. Formation rules define the structure of members of different syntactic categories by these sets. The formation rules are split in two parts, a part for Library and a part for Rules.

The following table 5.3, shows the formation rules.

$$
\begin{aligned}
S &= OPT \mid id = e; \mid If(e)\ S_1\ Else\ S_2 \mid If(e)\ S \mid ForeachFile\ S \mid ForeachFileIt\ S \mid Selected; \\
&\quad \mid Deselected; \mid S_1\ S_2 \mid \{\ S\ \} \mid Rule\ id\{\ Select\ e; Do\ S_1, S_2, \ldots, S_n;\} \mid DCL \mid \epsilon \\
OPT &= WorkFolder\ p; \mid RunOption\ Recurring; \mid RunOption\ Once; \mid WorkInSubFolders\ b; \\
&\quad \mid Exclude\ p_1, \ldots, p_n; \mid RunRules\ id_1, \ldots, id_n;\ S \mid FinallyOnChange\ Message(s); \\
&\quad \mid FinallyOnChange\ Beep(); \\
do &= Move(e) \mid Copy(e) \mid Rename(e) \mid Delete() \\
DCL &= T\ id; \mid T\ id = e; \mid Calculator < T >\ id\ (T_1\ id_1, \ldots, T_n\ id_n)\ \{\ S\ \} \\
&\quad \mid Selector\ id\ (T_1\ id_1, \ldots, T_n\ id_n)\ \{\ S\ \}
\end{aligned}
$$

$$
\begin{aligned}
e &= e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \mid e_1\ \%\ e_2 \mid (e) \mid -e \mid e_1\ OR\ e_2 \mid e_1\ AND\ e_2 \\
&\quad \mid NOT\ e \mid e_1 == e_2 \mid e_1\ !\ = e_2 \mid e_1 < e_2 \mid e_1 <= e_2 \mid e_1 > e_2 \mid e_1 >= e_2 \mid id \\
&\quad \mid id\ (e_1, \ldots, e_n) \mid id\ (all_1, \ldots, all_n) \mid ThisFile(ms) \mid ThisFile.HashValue \\
&\quad \mid ThisFile.IsSelected \mid n \mid s \mid b \mid p \mid t \mid d \mid re
\end{aligned}
$$

Table 5.3. Formation rules.

The transition rules of the language are described in later sections. Not all elements of "*OPT*" in table 5.3 have formal definitions / transition rules in the following parts of the definition of PIMPLs semantics.

5.3.2 Environments

This section describes the environments and storage model of the SOS and some auxiliary functions.

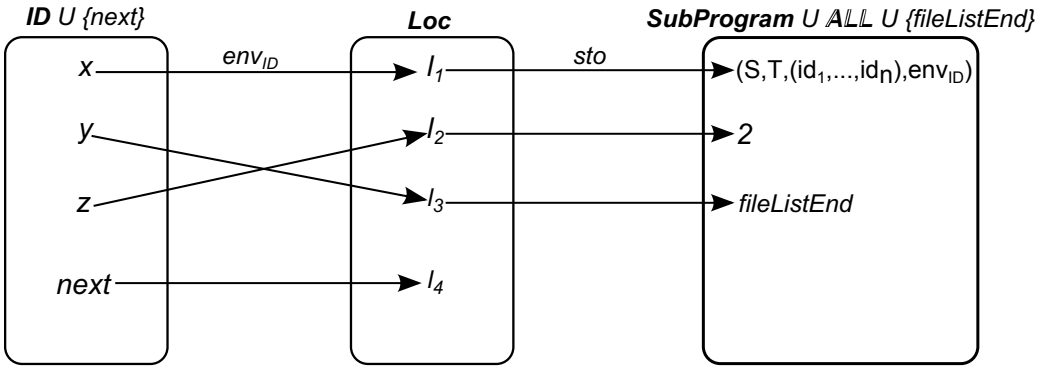


Figure 5.1. Graphical representation of how the partial function env_{ID} and *sto* works.

An illustration of the environment store model used in the SOS of PIMPL can be seen in figure 5.1. The figure is elaborated further in the following section.

ID environment

The ID environment is used to map identifiers, i.e., ID to locations. The identifiers contained in the environment covers the variables, rules and subprograms.

The set of ID environments is called **EnvID**, where env_{ID} *is an arbitrary member of* **EnvID**.

$$\textbf{EnvID} = ID \cup \{next\} \rightharpoonup \textbf{Loc}$$

Where "next" is a pointer that is always bound to the next available location in memory.

Loc is the set of locations, i.e. memory storage cells. We let l denote an arbirary element of **Loc**. In the following we assume that **Loc** = $\mathbb{N}$ since the memory cells can be index with natural numbers.

We shall assume the existence of a function *new*:

$$new : \textbf{Loc} \rightarrow \textbf{Loc}$$

that for every location $l \in$ **Loc** returns the next available location in memory.

Here follows the notations for updating ID environments. For ID environments we write env_{ID} [$id \mapsto l$], where $\mapsto$ reads "maps to", to denote the environment env'_{ID} that is:

$$env'_{ID}\ y = \begin{cases} env_{ID}\ y & if\ y \neq id \\ l & if\ y = id \end{cases}$$

Storage environment

In order to represent lists ids used in parameters, for subprograms, we let List<ID> denote a list of metavariables from the syntactic category ID: (id_1 ... id_n). We let () denote an empty list.

The set of ID environments is called **EnvID**, where $env_{ID} \in$ **EnvID**.

$$\textbf{Sto} = \textbf{Loc} \rightharpoonup SubProgram \cup \mathbb{ALL}$$

The storage partial function *sto* maps a location l to a subprogram declaration or an element of $\mathbb{ALL}$. It is a partial function because not all possible location l have a mapping defined.

When a location l maps to a subprogram, i.e. Selectors, Calculators, and Rules it maps a member of SubProgram. SubProgram is the set of subprograms defined by the following Cartesian product : **Stmts** $\times$ **Types** $\times$ $List < ID > \times$ **EnvID**.

Here follows the notations for updating the storage. For storages we write $sto[\ l \mapsto allv \mid (S,\ (id_1\ ...\ id_n),\ env_{ID})]$, where $\mid$ reads "or", to denote that a location l can map more than one thing and $\mapsto$ reads "maps to", to denote the environment sto' that is:

$$sto'\ y = \begin{cases} sto\ y & if\ y \neq l \\ (\ allv \mid (S,\ (id_1\ ...\ id_n),\ env_{ID})) & if\ y = l \end{cases}$$

File environment

Now follows the environment for files. There exists only one global file environment from the perspective of a PIMPL program. The file environment is called env_F. The file environment is an attempt to simulate the file system of a computer. This file environment contains mappings from paths to all files in the file system. A PIMPL program is constantly aware of the working set of files, which is defined by the WorkFolder-, WorkInSubFolders- and Exclude declarations in the start of a PIMPL rule file. This working set of files is a subset of all files in the file environment env_F.

$$env_F = pv \rightharpoonup file \tag{5.1}$$

Files is the set of files: $file \in$ **Files** and $file$ is a tuple that represents an abstraction for an actual file in a computers file system defined as follows.

The tuple $file$ contains four elements. The first element represents the path to the file in the file system. The second element $filedata$ is the binary content of the file. The third element is an *mstov* function. **MSTOV** is the set of partial functions that maps a Metadataspecifier value to metadata value, saved in a file or in the file system.

A *mstov* partial function is then a mapping to a specific file's metadata. In case there is no metadata for the given file, the *mstov* function maps to the default value. The fourth element *bv* is a boolean value which indicates whether the file is considered selected or not selected in the current Selector call.

We assume the existence of a *NextFile* partial functions which maps a file from the working set of files to the next file in the working set of files, which is a subset of all the files in the global file environment env_F. A *nextFile* partial function maps a file to $fileListEnd$ in case no next file exist in the working set.

$$\begin{aligned}
&\textbf{NextFile} : \textbf{Files} \cup \{firstFile\} \rightharpoonup \textbf{Files} \\
&\qquad nextFile \in \textbf{NextFile} \\
&\textbf{MSTOV} : \mathbb{MS} \rightharpoonup \mathbb{DIT} \cup \mathbb{S} \\
&\quad file = (pv,\ filedata,\ mstov,\ bv) \\
&\qquad file \in \textbf{Files},\ fileListEnd \in \textbf{Files} \\
&\qquad where\ mstov \in \textbf{MSTOV}.
\end{aligned}$$

Where $firstFile$ always maps to the first file of the working set in the current file environment. $firstFile$ maps to $fileListEnd$ in case there is no files in the working set.

For instance: The following syntax "ThisFile('Photo+CameraModel')" would result in a lookup using the partial function **MSTOV** of the current file, denoted by "ThisFile" in the language. The syntax "'Photo+CameraModel"' of the syntactic category MetaSpec gives an actual Metadataspecifier value msv. This msv is then used as an argument for the *mstov* partial function of the current file where the result would be a string containing the metadata for the specified metadata: $mstov(msv) = sv$ where sv could be a string such as "Nikon".

Here follows the notations for updating the environment. For file environments we write $env_F\ [\ pv\ \mapsto\ (pv,\ mstov,\ bv)\ ]$ to denote the environment $env_F^{''}$, which replaces the old global file environment env_F. $env_F^{''}$ is given by the following:

$$env_F^{'}\ y\ = \begin{cases} env_F\ y & if\ y\ \neq\ pv \\ (pv,\ filedata,\ mstov,\ bv) & if\ y\ =\ pv \end{cases}$$

$$env_F^{''}\ firstFile\ =\ env_F^{'}\ pv$$

We let $env_F\ [\ pv\ \mapsto \epsilon\ ]$ denote the environment $env_F^{''}$, which replaces the old global file environment env_F. $env_F^{''}$ is given by the following:

$$env_F^{'}\ y\ = env_F\ y\ \ if\ y\ \neq\ pv$$

$$env_F^{''}\ firstFile\ =\ nextFile(env_F\ pv)$$

Auxiliary functions

We assume the existence of a Default function. This Default function is used for variable declarations without initialization.

$$Default:\ T\ \longrightarrow\ allv$$

Table 4.2 shows the default values of each declarable type.

$$\mathcal{DITTOS}:\ ditv\ \longrightarrow\ sv$$

$$\mathbb{TOS}:\mathbb{DITV}\cup\mathbb{S}\cup\mathbb{P}\longrightarrow\ \mathbb{S} \qquad\qquad \mathcal{TOS}\in\mathbb{TOS}$$

$$\mathcal{STOREG}:\ sv\ \longrightarrow\ rev$$

$\mathcal{STOREG}$ is a function that maps strings of characters to a RegEx where the content of the input string is interpreted as a RegEx which is returned.

$\mathcal{DITTOS}$ is a function that maps a ditv, i.e. a *dv* or a *nv* or a *tv*, to an equivalent string value *sv*. $\mathcal{DITTOS}$ would for instance if given an integer 200 return a String containing the characters '2', '0' and '0'.

$\mathcal{TOS}$ is a function which maps a *ditv* or a *sv* or a *pv* to a string.

We also assume the existence of a *StmtExpand* function that takes a statement S and expands the statement to the set of terminal statements which can be derived from the composite statement formation rule $S \rightarrow S_1S_2$. A set containing statement S is returned in case statement S is not composite statement.

$$StmtExpand:\ S\ \longrightarrow\ Stmts$$

We assume the existence of a function *GetHashValue*, in order to handle ThisFile.HashValue language construct, which takes a pathvalue (*pv*) and maps it to a string value (*sv*) which

contains a string representation of a hashvalue in hexadecimal notation. This hashvalue is based on the content of the file.

$$GetHashValue : pv \longrightarrow sv$$

The files of the file environment needs to have their fourth element, the value that indicates if the file is selected or not, reset to $\perp$ (false) every time a PIMPL Rule has completed its execution. We assume the existence of a function *ResetFileEnvironment* that takes a file environment env_F ands returns a new file environment env'_F which is similar to env_F except that all files in the environment have their fourth element set to $\perp$.

$$ResetFileEnvironment : env_F \longrightarrow env'_F$$

A new *pv* which ends with a given file name must be created when a rename do operation has to be executed. An auxiliary function which handles exactly this is informally defined below:

$$ReplaceFileName : pv \times sv \longrightarrow pv$$

The *ReplacefileName* function takes a path *pv* and a string of characters *sv* as arguments and returns a new path. The file name at the end of the input path is removed and replaced with the given string value *sv* in the resulting path. If for instance the function was given a path "C:\users\user\document.doc" and a string "homework.doc" the returned path would be: "C:\users\user\homework.doc". If a file exists at the given path with the given name, then the resulting path is appended a unique number as a suffix, e.g. "C:\users\user\homework (2).doc".

When execution nested *ForeachFile* statements new implicit identifiers (id) are needed for each nesting level. It is assumed that there exists a function *CurrentFileName*, which maps an identifier environment (env_{ID}) and a storage (*sto*) to an identifier for an implicit variable for the current file.

$$CurrentFileName : env_{ID} \times sto \longrightarrow id$$

CurrentFileName uses an identifier *_implicitFileNestingLevel*, in the given env_{ID}, for a variable saved in the given storage. *CurrentFileName* is only defined if *_implicitFileNestingLevel* is present in the given identifier environment and storage. This variable is used to identify the current *ForeachFile* nesting level. It then uses this nesting level in order to create and return an identifer, for a path the current file, which matches the current nesting level.

5.3.3 Transition system

The transition systems of PIMPL are defined by the SOS. These transition systems are directed graphs, that represents possible transitions to and from different states. A transition system is a tuple (Γ, $\longrightarrow$, T), where:

- Γ is a set of states - the vertices
- $\longrightarrow \subseteq \Gamma x \Gamma$ is the transition relation - the edges

- $T \subseteq \Gamma$ is the set of terminal states

The following describes the transition rules for expressions.

$$\Gamma = Exp \cup \mathbb{ALL}$$
$$\longrightarrow_e \subseteq \Gamma \times \Gamma$$
$$T = \mathbb{ALL}$$

The $\longrightarrow_e$ is defined by the transition rules in section G.

The format of the transition rules for expressions is as following:

$$env_{ID}, env_F \vdash e \rightarrow_e allv$$

The following describes the transition system for statements in PIMPL.

$$\Gamma = (\textbf{Stmts} \times \textbf{EnvID} \times \textbf{Sto} \times \{env_F\}) \cup (\textbf{EnvID} \times \textbf{Sto} \times \{env_F\})$$
$$\longrightarrow_s \subseteq \Gamma \times \Gamma$$
$$T = \textbf{EnvID} \times \textbf{Sto} \times \{env_F\}$$

The $\longrightarrow_s$ is defined by the rules in section G.

Γ is a union between the two Cartesian products. The first Cartesian product results in a set of ordered sets where the first element is a statement and the second is an ID environment env_{ID} and the third element is the global file environment env_F. The second Cartesian product results in a set of ordered sets where the first element is an ID environment and the second element is the global file environment env_F. So Γ includes states with statements and states without statements i.e. terminal states. An ID environment and the global file environment is present in all states since statements will manipulate these two environments.

The format of the transition rules for statements is as following:

$$\langle S, env_{ID}, sto, env_F \rangle \rightarrow_s (env'_{ID}, sto', env'_F)$$

Transition rules

In this section a part of the transition rules are presented. Throughout the section the focus is on how PIMPL work with files. First there is a short informal description of the WorkFolder statements, to give an understanding of the initializing of the file environment. The remaining transition rules can be seen in appendix G.

The WorkFolder statements includes *WorkFolder*, *WorkInSubFolders*, and *Exclude*. The WorkFolder statements are described informally because they are unnecessary for the understanding of how a PIMPL program handles files. They are merely a way to create a list of files for the PIMPL program to work on. They are also highly dependent on the actual file system in the operating system and this is not the focus of the SOS.

Afterwards *ForeachFile* is presented, including the use of ThisFile(ms) inside it, these are relevant in context of Rules and the selection of files. Then Rule and RunRules can be

described, as Rules uses the ForeachFile to run through the files. Rules can be elaborated with a description of Selector declaration and call, and the Do operation Move.

WorkFolder: *WorkFolder* is a statement which must be followed by an absolute path to a folder in the file system. From this folder it creates a list of files. These files are used to redefine *NextFile*. *NextFile* now returns the first file of the list, given $firstFile$ as input.

NextFile now maps a given file from the list to the next file in the list, except the last file. When given the last file *NextFile* maps to $fileListEnd$.

WorkInSubFolders is a statement that, if followed by *True* or *Yes*, updates the mappings of *NextFile* to include the files in the subfolders of the *WorkFolder* path.

Exclude is a statement that again updates the mappings of *NextFile* to exclude a set of files given by one or more paths to folders or files.

ForeachFile: The transition rules for the *ForeachFile* statement can be seen in table 5.4. There are five separate transition rules, namely [FFILE-F-S], [FFILE-S], [FFILE-I], [FFILE-EMPTY], and [FFILE-END].

[FFILE-F-S] is a start transition for an outermost *ForeachFile* statement, if the *ForeachFile* statement contains nested *ForeachFile* statements. Some new environments must be initialized before execution of the statement (S). These updates are reflected in the where side conditions.

The first *where condition* looks up the first file in the file environment env_F. The second *where condition* introduces a new implicit identifier, *_implicitFileNestingLevel* , which maps to location l', in a new identifier environment env''_{ID}. The identifier *_implicitFileNestingLevel* must not already be defined in env_{ID}. Next a new storage sto[4] which maps l' to the number 1 is defined, which marks this *ForeachFile* statement as the outermost *ForeachFile* statement.

The fourth *where condition* introduces a new identifier *id* which is equal to the return value of the *CurrentFileName* function. The returned identifier is based on the given identifier environment and storage, which contains the current *ForeachFile* nesting level. The identifier *id* is then mapped to a new location l in a new identifier environment env'_{ID}. l is then mapped to the path of the first file ($firstFilePath$) given by the first *where condition*.

The first premise of [FFILE-F-S] says that the statement S of the *ForeachFile* statement must be executed in the environments env'_{ID}, sto'', *and* env_F where the notion of the current file is present, which in this case is the first file, given by the side conditions.

The *ForeachFile* statement loops over the remaining files in the second premise of [FFILE-F-S] using the two transition rules [FFILE-I] and [FFILE-E].

[FFILE-S] is the start transition for a *ForeachFile* statement when the *ForeachFile* statement is an inner *ForeachFile* statement, i.e. there is already an implicit variable *_implicitFileNestingLevel* defined in the current environments. The [FFILE-S] transition rule does the same as [FFILE-F-S] except it does not initialize the *_implicitFileNestingLevel* variable but it increments it by 1.

	$\langle S, env'_{ID},\ sto'',\ env_F\rangle \rightarrow_s \langle env'_{ID},\ sto^3,\ env''_F\rangle$ $\langle ForeachFileIt\ S,\ env''_{ID},\ sto^3,\ env''_F\rangle \rightarrow_s \langle env''_{ID},\ sto',\ env'_F\rangle$ $\langle ForeachFile\ S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto',\ env'_F\rangle$
$[FFILE - F - S]$	$where\ nextFile(firstFile) = (firstFilePath,\ filedata,\ mstov,\ bv)$ $and\ env''_{ID} = env_{ID}[\_implicitFileNestingLevel \mapsto l']$ $and\ sto^4 = sto[l' \mapsto 1]$ $and\ id = CurrentFileName(env''_{ID}, sto^4)$ $and\ env'_{ID} = env''_{ID}[id \mapsto l]$ $and\ sto'' = sto^4[l \mapsto firstFilePath]$ $if(\ env_{ID}(\_implicitFileNestingLevel) \uparrow {}^2)$ $and((\ env_{ID}(id) \uparrow))$
	$\langle S, env'_{ID},\ sto'',\ env_F\rangle \rightarrow_s \langle env'_{ID},\ sto^3,\ env''_F\rangle$ $\langle ForeachFileIt\ S,\ env''_{ID},\ sto^3,\ env''_F\rangle \rightarrow_s \langle env''_{ID},\ sto',\ env'_F\rangle$ $\langle ForeachFile\ S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto',\ env'_F\rangle$
$[FFILE - S]$	$where\ nextFile(firstFile) = (firstFilePath,\ filedata,\ mstov,\ bv)$ $and\ env''_{ID} = env_{ID}[\_implicitFileNestingLevel \mapsto l']$ $and\ sto^4 = sto[l' \mapsto sto(env_{ID}(\_implicitFileNestingLevel)) + 1]$ $and\ id = CurrentFileName(env''_{ID}, sto^4) and id \neq CurrentFileName(env_{ID}, sto)$ $and\ env'_{ID} = env''_{ID}[id \mapsto l]$ $and\ sto'' = sto^4[l \mapsto firstFilePath]$

Table 5.4. Big-step SOS for ForeachFile Starts.

The [FFILE-I] transition rule can be seen in table 5.5. The [FFILE-I] transition rule describes the remaining iterations of a *ForeachFile* statement. The *where conditions* sets up an identifier environment (env'_{ID}) and a storage (sto'') which contains the current file. The statement S is to be executed within these environments. The current file is based on the previous file's next file.

The [FFILE-END] transition rule concludes that the next file is the end of the file list ($fileListEnd$), which means there is no more files to iterate over. This concludes a *ForEachFile* statement and results in the state given by the last iteration.

The [FFILE-EMPTY] transition rule concludes that the file list is empty and that the *ForeachFile* statment should simply result in no state changes.

[2] "func(b)↑" is understood as "f(b) is undefined"

$$\frac{\begin{array}{l}\langle S, env'_{ID},\ sto'',\ env_F\rangle \rightarrow_s \langle env'_{ID},\ sto^3,\ env''_F\rangle \\ \langle ForeachFileIt\ \ S,\ \ env'_{ID},\ \ sto^3,\ \ env''_F\rangle \ \ \rightarrow_s \\ \langle env'_{ID},\ sto',\ env'_F\rangle\end{array}}{\langle ForeachFileIt\ S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto',\ env'_F\rangle}$$

$[FFILE - I]$

$$\begin{array}{l} where\ id = CurrentFileName(env_{ID}, sto) \\ and\ nextFile(env_F(sto(env_{ID}(id)))) = (currentFilePath,\ filedata,\ mstov,\ bv) \\ and\ env'_{ID} = env_{ID}[id \mapsto l] \\ and\ sto'' =\ sto[l \mapsto currentFilePath] \end{array}$$

$[FFILE - EMPTY]$

$$\langle ForeachFile\ S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto,\ env_F\rangle$$

$$if(nextFile(firstFile) = fileListEnd)$$

$[FFILE - END]$

$$\langle ForeachFileIt\ S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto,\ env_F\rangle$$

$$if(\ nextFile(env_F(sto(env_{ID}(CurrentFileName(env_{ID}, sto))))) = fileListEnd)$$

Table 5.5. Big-step SOS for ForeachFile iterative and end.

ThisFile(ms): The transition rule for *ThisFile*(*ms*), [TS-MS], can be seen in table 5.6. It is an expression, and it results in a value based on the metadata from the current file.

On the premise that the metadata specifier, *ms*, evaluates to an actual value, *msv*, it can be concluded that *ThisFile*(*ms*) evaluates to an actual value, *allv*.

This requires that the current file path is equal to *pv*, as seen in the first side condition. Where *pv* in the second side condition should be equal to a file with the function *mstov*. This function maps between a metadata specifier value, *msv*, to a value, *allv*, that is saved in the metadata for the file. In the third side condition, *allv*, which is the result of *ThisFile*(*ms*), should be equal to the result of the function *mstov* with the parameter *msv*.

	$$\frac{ms \rightarrow_e msv}{env_{ID},\ sto,\ env_F \vdash ThisFile(ms) \rightarrow_e allv}$$
$[TF - MS]$	*where* $sto(env_{ID}\ \_filepath) = pv$ *and* $env_F\ pv = (pv,\ filedata,\ mstov,\ bv)$ *and* $allv = mstov(msv)$

Table 5.6. Big-step SOS for ThisFile(ms).

Rule: A *Rule* is generally a wrapping of a *ForeachFile* with an if-statement, where operations should be executed on the chosen files. *Rule*, [DCL - R], can be seen in table 5.7.

In detail the execution of a *Rule* declaration, results in an update in the id environment, env_{ID}, and the *sto*. In the new env_{ID}, the rule's *id* maps to the location *l*. In the *sto* the location *l* maps to a statement *S* with the type *ok*, zero parameters and the current env_{ID}.

The side condition defines the statement *S* to be a *ForeachFile*, with an if-statement with the boolean condition *e*, and a body that contains the Do operations S_1 to S_n.

RunRules: Here follows an explanation of the [RUNR] transition rule, for *RunRules*, seen in table 5.7. Note that a *RunRules* call is a statement, because it changes the state of the storage and the file environment. *RunRules* must be executed after all declarations have been made. This is reflected in the [RUNR] transition rule's premise where S_{Dcls} is evaluated before S_{Rules}. Where S_{Dcls} is the statements preceding *RunRules* and S_{Rules} is a combination of the statements from the Rules' $(id_1, ..., id_n)$ bodies.

A *RunRules* statement will be implicitly declared in case no *RunRules* is declared in a PIMPL program. The references to rules $(id_1...id_n)$ of this implicitly declared *RunRules* comprises of all the identifiers for all declared rules in the textually defined order.

$$\langle Rule\ id\{Select\ e; Do\ S_1, S_2, \ ... \ , \ S_n;\}, \ env_{ID}, \ sto, \ env_F\rangle \longrightarrow_S (env_{ID}[\ id \mapsto l], \ sto[l \mapsto (S, \ ok, \ (), \ env_{ID})], \ env_F)$$

$[DCL - R]$

$$where\ S = ForeachFile\ If(e)\{\ S_1\ S_2\ \ ...\ \ S_n\}$$

$$\frac{\langle S_{Dcls}, \ env_{ID}, \ sto, \ env_F\rangle \rightarrow_s \langle env'_{ID}, \ sto', \ env_F\rangle \quad \langle S_{Rules}, \ env'_{ID}, \ sto', \ env_F\rangle \rightarrow_s \langle env'_{ID}, \ sto'', \ env'_F\rangle}{\langle RunRules\ \ id_1, \ ... \ , id_n; \ S_{Dcls}, \ env_{ID}, \ sto, \ env_F\rangle \rightarrow_s \langle env'_{ID}, \ sto'', \ env'_F\rangle}$$

$[RUNR]$

$$where\ (sto'(env'_{ID}\ id_1) = (S_1, \ ok, \ (), \ env^1_{ID})) \ ... \ (sto(env_{ID}\ id_n) = (S_n, \ ok, \ (), \ env^n_{ID}))$$

$$and\ S_{Rules} = \ S_1\ S_2\ ...\ S_n$$

Table 5.7. Big-step SOS for Rule and RunRules.

Selector declaration: The transition rule for selector declarations [DCL-S] can be seen in table 5.8. The transition rule is an axiom which adds a new subprogram to the location l and updates an env_{ID} to map to this location.

Selector call: Here follows an explanation of the [CALL − S] transition rule, for Selector calls, in table 5.8. Note that a Selector call is an expression, because it results in a value.

The first premise requires that all the syntactic values of the arguments to the Selector can be converted to actual values.

In the second premise, for the environments env_{ID} *and* env_F a statement transition is performed where statement S is executed and environment env'_{ID} is updated to env''_{ID} and env_F is updated to env'_F. env_F is updated because the Selector might change the value in the current file tuple which indicates whether the file is selected or not.

The update of env'_{ID} is a creation of a new scope, and it includes creation of three different local variables for this new scope. First a variable *_IsSelected* is created and initialized to the default Selector return value: $\perp$ (false). Secondly a variable $\_filepath$ is created and initialized to the current file's path which is found in env_{ID}. Lastly variables for all the parameters are created, and the variables are initialized to the parameters actual values. The statement S, in the second premise, originates from a lookup in the env_{ID} environment, as it can be seen in the first side condition.

_IsSelected is then evaluated to the boolean value *bv* in the environment env''_{ID} from the resulting state of the transition. This boolean value is the result of the Selector call, which is reflected in the conclusion of the transition rule.

At the end, in side conditions two, env'_F is updated with new information in the current file, here *_IsSelected* is set to *bv*, i.e. if the file was selected.

$$[DCL-S] \quad \langle Selector\ id(T_1\ id_1,\ ...\ T_n\ id_n)\{\ S\ \},\ env_{ID},\ sto,\ env_F\rangle \longrightarrow_S (env_{ID}[\ id \mapsto l],\ sto[l \mapsto (S, Bool,\ (id_1,\ ...\ , id_n),\ env_{ID})]\ env_F)$$

$$[CALL-S] \quad \frac{\begin{array}{l} all_1 \rightarrow_e allv_1,\ ...\ , all_n \rightarrow_e allv_n \\ \langle S, env'_{ID}[\_IsSelected \mapsto l',\ \_filepath \mapsto l'',\ id_1 \mapsto l_1,\ ...\ ,\ id_n \mapsto l_n],\ sto[l' \mapsto \bot,\ l'' \mapsto currentFilePath,\ l_1 \mapsto allv_1,\ ...\ ,\ l_n \mapsto allv_n],\ env_F\rangle \rightarrow_s \langle env''_{ID}, sto',\ env_F\rangle \\ env''_{ID},\ sto' \vdash \_IsSelected \rightarrow_e bv \end{array}}{env_{ID},\ sto,\ env_F \vdash id(all_1,\ ...\ , all_n) \rightarrow_e bv}$$

$$where\ env_{ID}\ id = l$$
$$and\ sto\ l = (S, Bool,\ (id_1,\ ...\ , id_n),\ env'_{ID})$$
$$and\ sto(\ env_{ID}(\_filepath)) = (currentFilePath,\ filedata,\ mstov,\ bv_{old})$$

Table 5.8. Big-step SOS for Selector declaration and Selector call.

Move: The statement *Move*(*e*), means to change the path of a file. It can be seen in table 5.9.

Move(*e*) results in an update of the file environment to env'_F, if the premise, that *e* evaluates to a path value, pv_1, is fulfilled.

Overall the side conditions defines how the new environment env'_F should be, on the condition that the current file is an actual file, i.e. it is not the end of the file list, $fileListEnd$. The first side condition "saves" the current files components, so that it can be modified. The second side condition removes the old file path from the file environment, by assigning the value ϵ. The third side condition then saves the old files components with the new file path, pv_1.

The other Do functions is similar to Move.

$$[DO-MOVE] \quad \frac{e \rightarrow_e pv_1}{\langle Move(e),\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto,\ env'_F\rangle}$$

$$where\ env_F(sto(env_{ID}(\_filePath))) = (pv_2,\ filedata,\ mstov,\ bv)$$
$$and\ env''_F = env_F[sto(env_{ID}(\_filePath)) \mapsto \epsilon]$$
$$and\ env'_F = env''_F[pv_1 \mapsto (pv_1,\ filedata,\ mstov,\ bv)]$$
$$if\ sto(env_{ID}(\_filepath)) \neq fileListEnd$$

Table 5.9. Big-step SOS for the Do function Move.

5.4 PIMPL's Type system

The type system for PIMPL includes extra abstract syntax, environments for types and type rules. This section cover these subjects.

5.4.1 Abstract syntax

The abstract syntax for types is an expansion of the abstract syntax in the SOS of PIMPL. In this section more syntactic categories for types and more formations rules to describe these are added.

Syntactic categories

Presented here is the additional syntactic categories for types in PIMPL, where T is a metavariable in the category **Types**. The syntactic categories can be seen in table 5.10.

$$
\begin{aligned}
T &\in Types \\
B &\in BaseTypes \\
NT &\in NumericTypes
\end{aligned}
$$

Table 5.10. Syntactic categories of types.

Types is a category that contains all types in PIMPL, including "Well typed" (*ok*). "Well typed" is a type that can be used to mark language constructions, which does not have a PIMPL type, e.g. If-statements, correct. **BaseTypes** consist of all the types in PIMPL where the **NumericTypes** only consist of the types Int, Date and Time, i.e. the types that can be used with e.g. the greater than operator(or similar comparison operators). The **NumericTypes** category is defined because these types can be used in similar situations, this reduces the amount of type rules.

Formation rules

The formation rules describes the actual connection between the syntactic categories and the types in PIMPL. The table 5.11, shows the formation rules for the type system.

$$
\begin{aligned}
NT &= Int \mid Time \mid Date \\
B &= String \mid Bool \mid Path \mid RegEx \mid NT \\
T &= B \mid ok
\end{aligned}
$$

Table 5.11. Formation rules for type system.

5.4.2 Environment

Now follows the environment for the types and a auxiliary function. In order to represent lists of types, from the set **Types** used in parameters for subprograms, we

let *List* < **Types** > denote a list of types: $(T_1 \ldots T_n)$. We let () denote an empty list. The type environment is a partial function:

$$E \ : \ ID \ \rightharpoonup \ \textbf{Types} \cup \textbf{SubProgramTypes}$$

Here **SubProgramTypes** is the set of subprograms defined by: ({*Calc*, *Rule*, *Selector*} × **Types** × *List* < **Types** >). SPT is a metavariable for elements of **SubProgramTypes**.

The type environment is a partial function because not all identifiers (*id*) maps to a type (*T*). *Calc*, *Rule*, *and Selector* are used to indicate if the subprogram is a Calculator, a Rule or a Selector respectively.

Here follows the notations for updating the environment. For type environments we now write $E\,[\ id \ \mapsto \ T\ ]$ to denote the environment E' that is given by the following

$$E'\ (y) \ = \begin{cases} E\,(y) & if\ y \ \neq \ id \\ T & if\ y \ = \ id \end{cases}$$

and we write $E\,[\ id \ \mapsto \ SPT\ ]$ to denote the environment E'' that is given by

$$E''\ (y) \ = \begin{cases} E\,(y) & if\ y \ \neq \ id \\ SPT & if\ y \ = \ id \end{cases}$$

Auxiliary function

We assume the existence of a partial function *MetaDataType* which maps Metadataspecifier values (msv) to a Type T.

$$MetaDataType \ : \ msv \ \rightharpoonup \ T$$

MetaDataType is defined with a file "Metadata mapping to PIMPL type.txt", that can be found on appendix I. The file "pimplmetamap.map" includes multiple lines where each line describes a mapping from a Metadataspecifier value (msv) to a Type (T).

An example of such a line looks like this:

DateModified:Date

This line maps the Metadataspecifier value "$DateModified$" to the Type *Date*.

5.4.3 Type rules

Here follows an example of a type rule of PIMPL. The remaining type rules can be found in appendix G.

$[IF_{STM}]$	$\dfrac{E \vdash e : Bool \qquad E \vdash S : ok}{E \vdash If(e)\ S : ok}$
$[CALC_{DEC}]$	$\dfrac{E \vdash S_1 : ok \qquad E[id \mapsto (Calculator, T, (T_1 \ldots T_n))] \vdash S_2 : ok}{E \vdash \ Calc < T > id(T_1\ id_1, \ \ldots\ , T_n\ id_n)\{\ S_1\ \}\ S_2 : ok}$

Table 5.12. Type rules for If statements and Calculator declarations.

The type rule for If statements [IF_{STM}] can be seen in table 5.12. For an If statement to be considered well typed (*ok*), the expression e, i.e. the boolean condition, must be of type *Bool* and the statement S, i.e. the body of the If statement, must be well typed.

A more advanced type rule for the *Calculator* declaration statements [$CALC_{DEC}$] can be also be seen in table 5.12. For a *Calculator* declaration statement to be considered well typed (*ok*), the body of the Calculator S_1 must be well typed. Secondly the statement preceding the *Calculator* declaration must be well typed within a new type environment updated in the second premise.

Implementation 6

The following chapter describes how the language processor for PIMPL is designed and implemented. As mentioned in section 3.4.2, the language processor is chosen to be a compiler. This chapter therefore starts with a presentation of a compilers phases. After the presentation follows a some theory and a description of each phase. The accompanying implementation of a PIMPL-to-C# compiler is in the following sections and chapters be referred to as "the compiler".

In short the compiler should take a source program written in the PIMPL language, and translate it into target code (C#). The three phases are: Syntax Analysis, Contextual Analysis and Code Generation, as shown in figure 6.1. In the first phase, the job is to scan the program and see if it is syntactically correct, according to the language grammar rules. Afterwards the code should be parsed to a data structure, which typically is an Abstract Syntax Tree (AST). The AST is then used in the Contextual Analysis phase. In this phase the job is to analyze if the program conforms to the contextual rules of the language. The output of the Contextual Analysis is a decorated AST, which is used as input to the Code Generation phase. Here the task is to convert the decorated AST into target code.

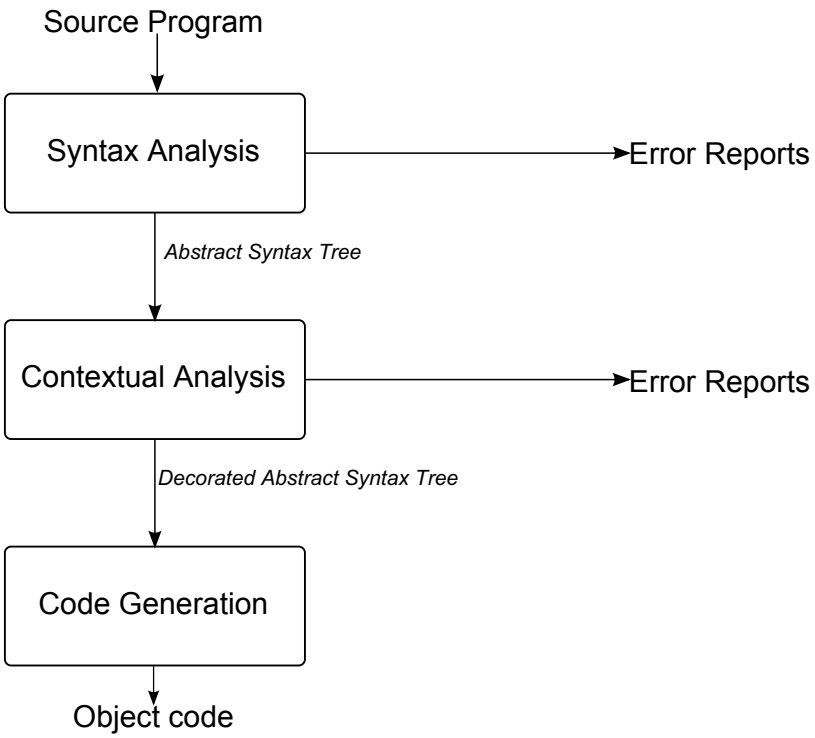


Figure 6.1. Overview of compiler phases.

A preprocessor is needed before a PIMPL program is compiled, in order to combine different parts of code (In PIMPL, this means combining Library-parts with the Rule-part). This is needed before the compiling process can begin. The Preprocessor for PIMPL is described before the actual compiler.

6.1 Preprocessor

Before the actual compilation, the preprocessor is invoked. The preprocessor used for PIMPL code files is tasked with merging PIMPL library files with the PIMPL rule program files. When a Rule program is compiled, the preprocessor copies all the Library code, of all the referenced library files into the start of a new file and then appends a copy of the Rule program. The output of this is a compilable combined file, that consist of all the Library-code followed by the Rule-code.

A Rule file and a Library file could also be connected using a linking process, where the compiled versions of both files are linked together, instead of using a preprocessor method to combined the files. This means that not all code will not have to recompiled every time code is changed. This saves time when compiling large code bases. It has been chosen to use a preprocessor, because PIMPL program source code files are not intended to contain more than a few hundred lines. As a consequence of the preprocessor, Library files have to be recompiled when a new Rule file is made, but this is of little significance as PIMPL programs consists of relatively small files.

6.1.1 Procedure

The preprocessor starts out by analyzing the given PIMPL rule file. It parses a list of referenced Library files using a regular expression. The new parsed list, of Library file names, is then used to lookup each of the files in the file system. It is important that the Library files is placed in the same folder as the Rule file. A new file is then created with the file extension ".cpimpl", which is an abbreviation for "Combined PIMPL". The content of all the Library files are then copied to this new file along with the content of the Rule file, except for the code for the Use declaration in the start of the Rule file. The Use declaration is only used for the preprocessor, and contains nothing of importance when these files has been combined.

6.1.2 Grammar changes

To make the rest of the compiler compatible with the code generated by the preprocessor, some changes have to be done to the CFG of PIMPL. In the grammar the Rule- and Library-part is set to be two different programs, and therefore needs to be compiled separately. With this approach to use Selectors in Rules, it is necessary to compile the Rules and Libraries as one file. Consequently the grammar needs to be changed in order to allow these two parts to be compiled together, i.e. Libraries should be compiled prior to the Rules.

The original EBNF grammar of PIMPL, shows how programs should be written, and the LL(1) implementation grammar is adjusted to enable parsing of the result of the preprocessor. The original EBNF grammar can be seen in listing 6.1, and the LL(1) implementation grammar in listing 6.2.

```
<Start> = (<LibraryPart> | <RulesPart>) $
```

Listing 6.1. EBNF grammar of the start of a PIMPL program

```
<Start>  = <StartB> $
<StartB> = <LibraryPart> <StartC> <RulesPart>
<StartC> = <LibraryPart> <StartC> | EPSILON
```

Listing 6.2. LL(1) grammar of the start of a PIMPL program

6.1.3 Line number mapping

A mapping between the original files line numbers and the line numbers of the new combined file is created in order to provide better error messages, with correct lines numbers, to the users. The preprocessor saves a mapping between the lines of the original files and the combined file. For example: A library starts on line 1 and ends on line 40, and the rule-part starts on line 41. This information is saved and can then be used to find the exact line number of an error, if the compiler finds one.

6.2 Syntax Analysis

In the syntax analysis phase of the compiler, the input is a program, i.e. source code, written in PIMPL and the output is an AST. This phase should both create the AST from the source code and look for syntactic errors. This section describes choices related to parser tools. A description of how the grammar is transformed to LL(1) is presented. This transformation is done because the chosen tool, JavaCC, is configured to only understand LL(1) grammars. Lastly follows a description of how the AST is designed and its creation implemented.

6.2.1 Theory

A Scanner takes a source program as input, and then analyzes a stream of characters. This stream of characters is then converted into a stream of Tokens, that is used as input to the Parser. Figure 6.2 shows an overview of the Syntax Analysis phase.

An AST is created along with the parsing of a program, this AST is roughly based on the CFG of the language. The AST is then modified using a visitor which transforms the tree. This AST is the result of the Syntax Analysis phase. Errors are reported if they are encountered during the execution of both the Scanner and Parser [Fischer et al., 2009].

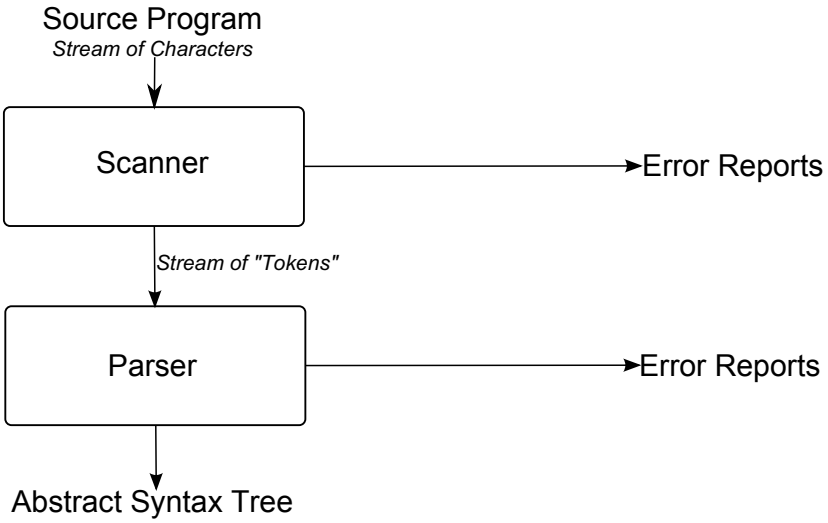


Figure 6.2. Syntax Analysis process.

Scanner

A lexical element, also called a token, is a character or a combination of characters which is allowed in a language. The tokens of PIMPL are defined with regular expressions. Appendix D, contains the Lexical Elements of PIMPL. When a program is scanned, the source program is converted into a stream of tokens. The Scanner reads the source program character by character, and tries to match these with the regular expressions for the tokens, to see if it is a correct element in the language. The stream of characters could for instance be “W-o-r-k-F-o-l-d-e-r” or “(”, which is matched respectively with

the Tokens "setWorkFolder" and "lparen". The character stream maintains the order of the source program, and if the characters cannot be matched with a Token, an error is reported [Fischer et al., 2009].

Parser

The input to a Parser is a stream of tokens acquired from the Scanner. The parser requests one token at the time and from here the parser decides which grammar productions that matches the token stream. It is possible for a Parser to look at more than one Token before it makes a decision. This is called the lookahead and this can be set to one or more Tokens. If the Parser cannot match the Token stream with a production from the CFG, an error is reported. It is possible to construct an AST while the Tokens are read and matched with grammar productions [Fischer et al., 2009].

As an example, see listing 6.3. If the Parser gets the Token "setWorkFolder", it knows that the grammar production needed is "<WorkFolderDef>". This is because the first element in this is "setWorkFolder", and it is the only grammar production with this exact Token as the first element token. It then concludes that the next two Tokens must be "pathConst" and "semiColon", or else the parsing terminates with an error. This means that the Parser has detected a syntax error.

```
<WorkFolderDef> = setWorkFolder  pathConst  semiColon
```

Listing 6.3. CFG production for the nonterminal WorkFolderDef.

6.2.2 Parsing strategy

Generally there are two types of tools based on the two different parsing methods: Top-Down parsing and Bottom-Up parsing, see figure 6.3 [Fischer et al., 2009] for a graphical representation of this.

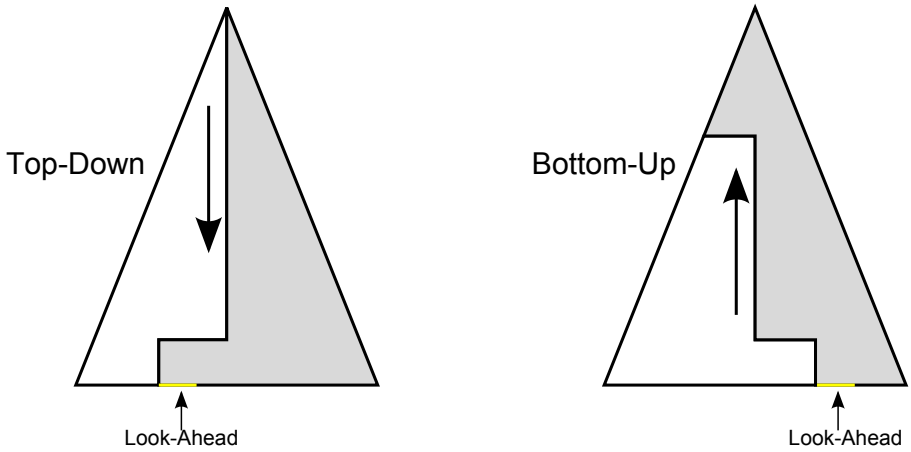


Figure 6.3. Top-Down and Bottom-Up parsing.

Top-Down parsers work on LL grammars, where LL is short for **L**eft-to-Right scanning and **L**eftmost derivation, and k is the number of tokens of lookahead. The Top-Down

parser looks at the root first, and then the leftmost child. If no child is present, the parent nodes next child is checked. Bottom-Up parsers work on LR grammars, and here LR is short for **L**eft-to-Right scanning and **R**ightmost derivation. The Bottom-Up parser starts at the leftmost child, and then works upwards and rightwards. The two parsing strategies does not work on the same class of languages, although there is a overlap, meaning that some CFG can be recognized by many strategies, which can be seen in figure 6.4. A Top-Down parser is easier to understand and implement, than a Bottom-Up parser, but the Bottom-Up parser works for a larger number of languages, as figure 6.4 is presenting. A difference is that Bottom-Up parsers can handle left recursion, while a Top-Down Parser cannot [Fischer et al., 2009].

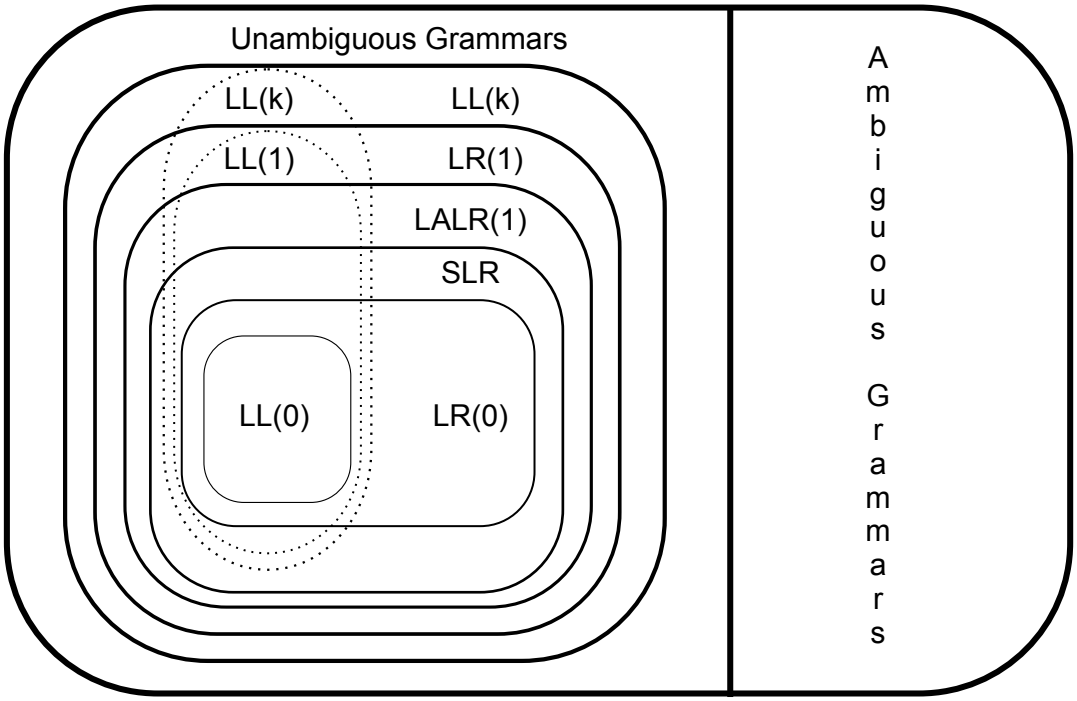


Figure 6.4. Graphical representation of which languages recognizes which grammars.

There are some benefits and disadvantages associated with both parsing strategies, and PIMPL is a language that can be parsed using both types although some modifications would have to be made to the original grammar.

The group has chosen a Top-Down parser because the parsing is more understandable than with the Bottom-Up parsers. A consequence of choosing a Top-Down parser is that more effort must be put into the design of the grammar in order to eliminate left recursion, thereby making the grammar LL(k) and still generate PIMPL.

There is a need to find a suitable tool for Top-Down parsing. The project group has been introduced to different compiler compiler tools during semester courses, and finds it convenient to use one of these tools.

6.2.3 Handwritten or tool-generated syntax analysis

When creating a scanner and parser, it is possible to construct it by hand or with the use of certain tools. A tool typically takes a CFG with lexical elements, and it then generates a scanner and/or parser for the grammar. It is also possible to make a scanner and parser by hand, but this is harder, more time consuming and error-prone, especially when making a Bottom-Up parser. When writing a special or small language, it can be better to create this by hand. Writing the scanner and parser by hand takes more time during development, but this can result in a more efficient scanner and parser, if the developers knows what they are doing. However compiler compiler tools works well and they make it faster to implement a scanner and a parser. A compiler compiler tool was therefore chosen to assist the implementation of a compiler [Fischer et al., 2009].

6.2.4 Scanning and Parsing tool

This subsection presents a discussion of parsing tools. At the end of the subsection parsing tool is chosen.

JavaCC

JavaCC is a scanner and parser generator which can be used for implementing parsers in Java. It generates a top-down recursive descent parser. JavaCC code can contain Java action code embedded in the JavaCC grammar. JavaCC does not create an AST structure, on its own, which can be used for the following compiler phases. Although with the help from the JJTree tool, JavaCC can generate actions for building the desired AST. JJTree transforms a JJT grammar to a JavaCC grammar which includes Java action code for buidling building an AST. JJTree generate the SimpleNode and an interface Node that provides methods for creating and manipulating the AST. This AST can then be traversed using a visitor pattern [JJTree, 2014].

ANTLR

ANTLR, like JavaCC, is a scanner and parser generator tool for Java, that allows LL(k) grammars and therefore uses a top-down table driven parsing strategy. ANTLR takes a grammar as input and generates the source code for a recognizer for that grammar. ANTLR also provides a tree structure, and an auto generated tree walker, that can be used to visit nodes. ANTLR has an dedicated IDE called ANTLRWorks, that provides automatically created token definitions, syntax highlighting and a comprehensive debugger [ANTLR, 2014].

Coco/R

Coco/R is a compiler generator that takes an attributed grammar of a source language and generates a scanner and parser for that language. It uses a top-down recursive descent strategy, and the accepted types of grammars are LL(k), as LL(1) conflicts can be resolved by semantic checks [Hanspeter Mössenböck and Markus Löberbauer and Albrecht Wöß, University of Linz, 2011].

Tool selection

The project group has chosen to use a tool for the generation of the scanner and parser components of the compiler. The chosen tool is JavaCC with JJTree. These tools was chosen because it was the tool the group liked the most, because of its easy syntax and setup time.

6.2.5 Transformation

This section contains an example of how the original PIMPL EBNF grammar is transformed into the needed LL(1) grammar used in the JavaCC tool. The original EBNF grammar is transformed into a BNF grammar, as some of EBNF's features for grouping and repetition are not allowed in BNF. The BNF transformation was used as an intermediate step between the original EBNF grammar and the desired LL(1) grammar. Appendix B shows the full version of the readable EBNF grammar. Appendix C shows the result of the transformation process from the original readable EBNF grammar to a final LL(1) implementation grammar.

The PIMPL language can be parsed using an LL(1) parser if the grammar is transformed. It is therefore not necessary to use an LL(k) parser. Consequently the original PIMPL grammar needs to be transformed into an LL(1) grammar. This transformation is described in the following sections.

The project group originally constructed an EBNF grammar, in order to define the PIMPL language. This original EBNF grammar was later transformed into BNF and then to LL(1). The EBNF was used to get a good overview of how the language should work. The final LL(1) implementation grammar has a problem, the dangling else problem. The dangling else is a problem where nested if-else statements can result in ambiguous grammar, as the else can cling to both if-statements. This problem is solved by letting an else clause be associated with the nearest if statement.

EBNF to BNF to LL(1)
Listing 6.4 shows how the production for **Values** is in the EBNF grammar. A new terminal EPSILON is needed, EPSILON defines a empty/null terminal. Here [...] denotes that the elements inside are optional, meaning this can be EPSILON, removing the need for a separate terminal for EPSILON. {...} is used when the elements inside can appear zero or more times.

```
1 <Values>               = [<Value>] { comma <Values> }
2
3 <LogicalOrExpression> = <LogicalAndExpression> | <LogicalOrExpression>
      logicalOrOperator <LogicalAndExpression>
```

Listing 6.4. Original **<Values>** production and **<LogicalOrExpression>**, in EBNF.

In order to transform the grammar from EBNF into BNF, these EBNF features must be eliminated and replaced by their corresponding BNF representation. Firstly, the repetition brackets ({...}) are removed by making this production into a new production, **<ValuesB>**, which contains the repeating part. Next the optional part ([...]) of the

<Values> production is removed by adding an EPSILON terminal to the production. Then these productions are separated with a (|) symbol. These three different productions represents that a **<Values>** nonterminal can result in either, many values separated by a comma, one value or no value at all. The result of the process is the BNF grammar shown in listing 6.5

```
1 <Values>               = <Value> <ValuesB> | <Value> | EPSILON
2 <ValuesB>              = comma <Values>
3
4 <LogicalOrExpression> = <LogicalAndExpression> | <LogicalOrExpression>
      logicalOrOperator <LogicalAndExpression>
```

Listing 6.5. Transformed **<Values>** production and **<LogicalOrExpression>**.

Now that the grammar is in BNF form, it can be transformed into LL(1) more easily. From BNF to LL(1), left factorization and left recursion must be eliminated. Left factorization must be eliminated as LL(1) only has one lookahead, meaning no production can start with the same nonterminal/terminal, as the grammar cannot decide between them.

An example of this is shown at line 1 in listing 6.5, where **<Values>** can be evaluated into **<Value> <ValuesB>** or **<Value>** or EPSILON. The two productions starts with the same nonterminal **<Value>** and an LL(1) parser can therefore not determine which production to choose by examining only the first nonterminal. This common left factor must be eliminated in order to make the grammar LL(1). This is done by combining these two productions into a new production **<ValuesB>**. The repetition is then moved into the <ValuesB> production, where this production can still result in a single **<Value>** or multiple **<Value>**. Other common left factors can be factored out in a similar way, and now left recursion must be eliminated.

In order to remove left recursion, the following was done: First, for each rule with the problem e.g. the production in line 4 in listing 6.5, introduce a new production and remove the left recursive production from where the problem was present. The nonterminals were changed to look like the nonterminals in line 4-5, in listing 6.6.

The result of the process can be shown in listing 6.6.

```
1 <Values>               = <Value> <ValuesB> | EPSILON
2 <ValuesB>              = comma <Value> <ValuesB> | EPSILON
3
4 <LogicalOrExpression>  = <LogicalAndExpression> <LogicalOrExpressionB>
5 <LogicalOrExpressionB> = logicalOrOperator <LogicalOrExpression> | EPSILON
```

Listing 6.6. LL(1) of the **<Values>**- and **<LogicalOrExpression>** production.

6.2.6 AST

JJTree generates Java action code, as mentioned in section 6.2.4. This code generates a tree structure associated with the compiled program, based on the CFG. By default this tree is a concrete syntax tree, where all non-terminals and terminals are nodes. The terminals

are, more precisely, the leaves of the tree. The productions in the grammar form the connections between the nodes parents and children in the tree.

This tree is needed in the next phases of the compiler, where this structure is traversed e.g. in order to type check the program. It is desirable to have a more abstract tree in the context of type checking and code generation, this reduces the amount of tree nodes which would have to be visited. The AST must still have comprehensive details to represent the input program, as the original code must be able to be recreated from the AST.

The tree generated by the syntax analysis is not entirely correct in terms of the expression nodes in the tree. The expression substrees in the AST are slightly malformed because of the way the gramma is build for expressions in PIMPL. To solve this problem an AST transformation visitor was introduced. The AST transformation visitor essentially transforms the expression subtrees from right oriented lists to a left oriented lists.

Design

It is helpful to design different parts of the tree separately, when designing the AST. The design of all the nodes can be seen on the CD in appendix I. In this sections the design of the IfStmt-node and the CalculatorDcl-node is presented and explained as examples.

Figure 6.5 illustrates the IfStmt-node. This node type can have a maximum of three possible children, a predicate, a true branch and a false branch. The false branch is optional. The predicate needs to be a Boolean expression, and this can also just be a Boolean value. The true and false branch are both Statements, they can for example be empty, or statements like a new IfStmt or an ForeachFile. There is no node called "Stmt", the child should just be one of the available Statements of PIMPL.

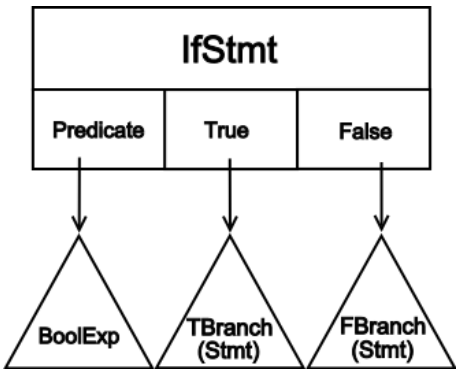


Figure 6.5. AST design for IfStmt-node.

In a CalculatorDcl node there is more information, saved in the node, than its children, see figure 6.6. Here it is necessary to hold information about the type and the ID of the Calculator. This information is placed in separate child nodes in the concrete parse tree. These are leaves and are only relevant for the CalculatorDcl, they therefore become a part of this node in the AST. Besides this information, the node contains some children. It contains a Formal parameter list and a number of Statements.

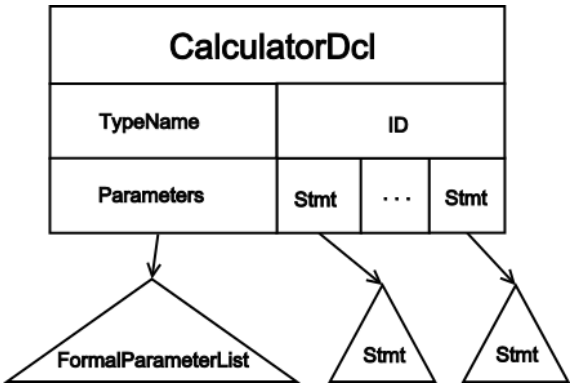


Figure 6.6. AST design for CalculatorDcl-node.

Implementation

The implementation of the AST is done using the JJTree tool, shown in 6.2.4. Basically the JJTree code dictates which non-terminals and terminals should generate a node, and what information these nodes should include in the different nodes.

For this there is a need to define an special node type, that can keep the information. This node class is called "PimplASTNode", and it inherits from the default node class "SimpleNode" which implements the interface "Node". The implementation of the interface "Node" is needed when JJTree generates the tree. A "PimplASTNode" has some fields to store the extra information, for example Value, ID and Type. In the action code associated to a node, the fields can be set to a value based on a Token. This can be a number from an Int or the characters from a String.

If a non-terminal should not create a node, it is marked with #*void*. Else it is marked with #*Name*, where Name is a meaningful name for the node. It is also possible to mark production in the non-terminal with #*Name*, this will create a node with the declared name if the production is taken.

Listing 6.7 shows the production for the non-terminal CalculatorDcl in JJTree code. On the first line #*CalculatorDcl* indicates the creation of a node with that name, and the action code in lines 2-5 creates two temporary Token variables, "type" and "id". In the actual production lines(7-9) these variables are assigned to a value from the input program. The first assignment is *id* = **<ID>** where **<ID>** is a terminal and a Token that is assigned to the variable. Here the Token includes the textual representation of ID of the declared Calculator. In *type* = *TypeName*() the variable, *type*, is assigned to the return value of the nonterminal *TypeName*(), which returns a Token, which includes the name of the Calculators return type. The action code for the production saves the temporary variables in the nearest declared node, in this case in the CalculatorDcl-node. "jjtThis" indicates this node. It is only the image of the token variable id, i.e. the textual representation of the id token, which is saved in the node. The entire Token is saved, in the case of the Token variable id, which includes line numbers.

```
void CalculatorDcl() #CalculatorDcl :
{
  Token type;
  Token id;
}
{
  < calculator > < langlebracket > type = TypeName() < ranglebracket >
  id = < ID > < lparen > FormalParameterList() < rparen >
  < lbrace > Stmts() < rbrace >
  {
    jjtThis.typeToken = type;
    jjtThis.ID = id.image;
  }
}
```

Listing 6.7. JJTree code for CalculatorDcl.

Another example non-terminal is the If-statement, which includes two productions. The first production indicates that no node should be made. This is because more information is needed in order to decide how many children the node should have. An if-node can have either two or tree children depending on the existence of a false branch. The first task is to check if a false branch exists and a temporary variable, containFalseBranch, is created. This variable is assigned to the return value from IfStmtB, which returns true if there is a false branch, i.e. if there exist a **<elseSymbol>** token in the input stream. This result is used to create the node with either two or tree children.

```
void IfStmt() #void :
{
  boolean containFalseBranch = false;
}
{
  < ifSymbol > < lparen > Expression() < rparen > Stmt()
  containFalseBranch = IfStmtB() #IfStmt( containFalseBranch ? 3 : 2 )
}

boolean IfStmtB() #void :
{}
{
  < elseSymbol > Stmt()
  {
    return true;
  }
|
  {
    return false;
  }
}
```

Listing 6.8. JJTree code for IfStmt.

6.3 Contextual Analysis

A CFG is not enough to describe the PIMPL language. Syntax rules are only enough to specify the format of well-formed programs. A set of contextual rules, must be checked in order to ensure that a given program is a valid PIMPL program. This section describes how PIMPL programs are validated according to the contextual constraints of PIMPL.

6.3.1 Choice of visitor pattern for the compiler

A way of traversing an AST must be chosen in order to decide if a given AST, which represents a program, conforms to the contextual rules of PIMPL. There are different ways to traverse an AST and some of them are described in this section. Lastly one of these are chosen to be the method used for the contextual analysis part of the compiler and later for the code generation part of the compiler.

The object oriented approach dictates that a new method must be added to each node class for every type of traversal the AST must support. This approach is very intuitive if the developer has experience with object-oriented programming languages. Each new traversal will be placed in many different class files. This results in scattered code.

The procedural method uses switch statements for traversing the AST. The code for each traversal in this approach is, unlike the object oriented approach, kept in one place. Depending on the AST this might cause the programmer to write more code than with other approaches.

A Visitor approach makes it easy to evoke different methods on the nodes in the AST. A visitor pattern allows methods to be encapsulated in a visitor class, instead of having each node class implement its own methods.

The project group decided that a visitor approach would be the best solution. Many different types of visitors exists, but the project group decided to use a variation of the Visitor pattern which exploits static overloading, because the chosen parser generator tool, JavaCC, provides a Java interface that follows this static overloading visitor approach.

The PIMPL compiler uses three different visitors, one for AST transformation, one for contextual analysis, and one for code generation.

The AST transformation visitor is a dynamic dispatch visitor variation which uses a single method to the apply its transformation, to the AST, to a Select group of Nodes.

6.3.2 Type checking

Type checking is done in order to enforce the rules of the type system of the programming language. The type rules are in turn designed to prevent runtime errors. Type rules check the expected types of arguments, for both subprograms and operators, and specify what the resulting type should be. This check is done by utilizing a visitor, that traverses the trees nodes. An example of a visit method in the contextual analysis visitor of the PIMPL compiler is shown in listing 6.9.

When the visitor reaches an IntConst node during the traversal of the AST, it will evoke a method that tries to parse an Int from the string in value in the node, and report an error if parseInt throws an exception.

```
@Override
public PimplType visit(final IntConst node, final Object data)
{
        try
        {
                Integer.parseInt((String) node.getValue());
                return node.setType(PimplType.intType);
        }
        catch (Exception e)
        {
                errorLog.reportError("Integer does not fit within a 32 bit
                    integer", node);
                return node.setType(PimplType.error);
        }
}
```

Listing 6.9. Type checking for IntConst.

6.3.3 Scope rules

The scope rules, as defined by the semantics in appendix G, must be enforced by the contextual analysis. It must be checked if a variable or subprogram, is declared before it is used. PIMPL Rules are an exception to this, since a rule can be referenced in the "RunRules" language construct in top of a ".pimpl" Rule-file, before the rule is actually declared. The contextual analysis still checks if the rules are actually declared, it simply postpones the analysis of RunRules until it has reached the end of the analysis as it can be seen in listing 6.10 and 6.11.

```
@Override
public Type visit(final RunRules node, final Object data)
{
    this.postponedRunRulesNode = node;

    return Type.correct;
}
```

Listing 6.10. Visitor method for RunRules nodes.

The code in listing 6.10 shows that the visitor method simply saves a global reference to the RunRules node. This node is later used in the visitor for Start nodes, as it can be seen in listing 6.11, and then concludes that the RunRules node is correct. The null check of the reference to the RunRules node in listing 6.11 exist because the RunRules is optional in a PIMPL program. The rules are simply evaluated in the order they were declared if RunRules is not defined.

```
@Override
public PimplType visit(final Start node, final Object data)
{
        final PimplType childType =
            node.setType(checkChildrensTypeCorrectness(node));

        // It does not make sense to perform contextual analysis on the
            RunRules node before the ruledcls have been visited

        // RunRules is optional
        if (postponedRunRulesNode != null)
        {
                postponedRunRulesNode.setType(checkChildrensTypeCorrectness
                (postponedRunRulesNode));

                if (postponedRunRulesNode.getType() == PimplType.error)
                {
                        errorLog.reportError("RunRules contains invalid rule
                            references", postponedRunRulesNode);
                        return node.setType(PimplType.error);
                }
        }

        return node.setType(childType);
}
```

Listing 6.11. Visitor method for Start nodes.

Symbol table

A symbol table is used to keep track of the symbols, and information about the symbols, used in the program that is undergoing compilation. Thus a symbol table takes part in enforcing the scope rules of the language. The symbol table used for the compiler has two main purposes. The first purpose is to keep track of which symbols are already declared in current and preceding scopes, in order to prevent multiple declarations using the same symbol in the same scope. The second purpose of the symbol table is to record and make available the type of the symbols as they are declared.

A program written in PIMPL could for instance declare a variable of type Int with the symbol "counter". The symbol table would then, upon a request, check if the symbol "counter" already exists for the current scope. If it does it would let the requesting code know that the symbol is already in use for the current scope. If the variable does not exist, then the compilation proceeds to make and new entry in the table, under the current scope, containing information about where in the code the new symbol was declared and what type associated with this new symbol in the future is.

Symbol table - concrete representation

The concrete representation of the symbol table used in the implementation of the compiler consists primarily of two data structures. A main data structure declared as seen in listing 6.12 and graphical represented in figure 6.7. This main data structure is a hash map which associates String objects, i.e. the textual representations of the symbols with stacks of SymInfo objects. A hash map has been chosen because it is an effective way to map Strings to objects. SymInfo objects contains information about a given symbol. The idea here is that the top of the stack, which is associated with a given String object

which in turn again is the textual representation of a symbol, is a SymInfo object that corresponds to the innermost declaration, which should be visible for the current scope, that makes use of this symbol table.

```
private final HashMap<String, Stack<ISymInfo>> symbolTableDictionary = new
    HashMap<String, Stack<ISymInfo>>();
```

Listing 6.12. Main data structure for the symbol table.

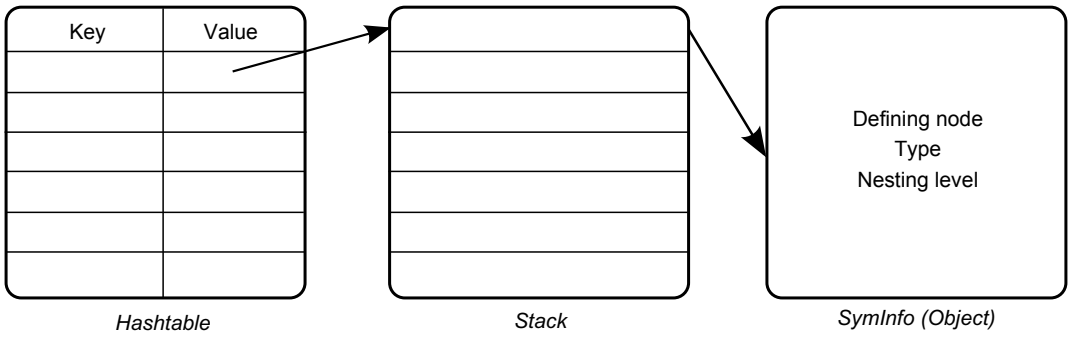


Figure 6.7. Graphical representation of the main data structure for symbol table.

A second helper data structure is used to keep track of all the symbols declared for each nesting level. This helper structure is declared as seen in listing 6.13 and is graphically represented in figure 6.8. The number of nesting levels in PIMPL are restricted in same way the C# langauge restricts its nesting level, and as there is no upper limit in C#. This second helper is a stack of Lists of String objects. A new list of String objects is pushed on the stack every time a new scope is opened. The topmost list is popped from the stack once a scope is closed. The list which is popped of the stack once a scope is closed, is used to update the main data structure by looking up each String object in the hash map and popping the stack accordingly, thereby removing a symbol info from the stack. This ensures that no symbols declared in the scope that was just closed persists in the symbol table.

This data structure allows the symbol table to check if an attempt to create a new entry for an identifier, that is already declared in the current scope, should cause an error or if it should simply hide a declaration from an outer scope. It also ensures that the symbol table is free of symbols declared in previously deeper nested scopes.

```
private final Stack<ArrayList<String>> introducedSymbols = new
    Stack<ArrayList<String>>();
```

Listing 6.13. Helper data structure for symbol table.

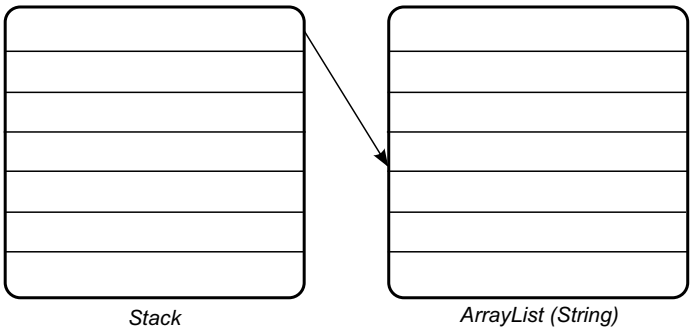


Figure 6.8. Graphical representation of the helper data structure for symbol table.

6.3.4 Decorated AST

The result of the contextual analysis, described in this section, is an AST decorated with types. These types can be concrete types found in the language and special types, like PimplType.error and PimplType.correct, which indicates whether or not the code that the AST represents conforms to the contextual constrains of PIMPL. The contextual correctness of the AST is reflected by nodes that would otherwise not be considered to have a type, such as nodes that represents statements. A subtree is considered correct if the type of the root node is not of type PimplType.error. If and only if the Start node, i.e. the root node of the AST, is marked as correct will the entire AST be considered contextually correct. This also means that the program conforms to the contextual constraints of PIMPL.

6.4 Code generation

The code generation phase of the PIMPL compiler is responsible for the generation of target code, based on a type decorated AST, which is the intermediate representation of a PIMPL program, which is the result from the previous phases. The code generation of PIMPL is done in a single parse using one visitor.

The code generator utilizes the visitor pattern as described in 6.3.1. Target code is then emitted to an output file for each node of the visited type decorated AST.

The rest of this section includes a description of the runtime environment of PIMPL and some problems, with accompanying solutions, that arose during the implementation of the translation from PIMPL to C#. The section does not describe how the C# is translated by the Visual C# Command Line Compiler. But the C# code is translated as shown in the tombstone diagram in figure 6.9.

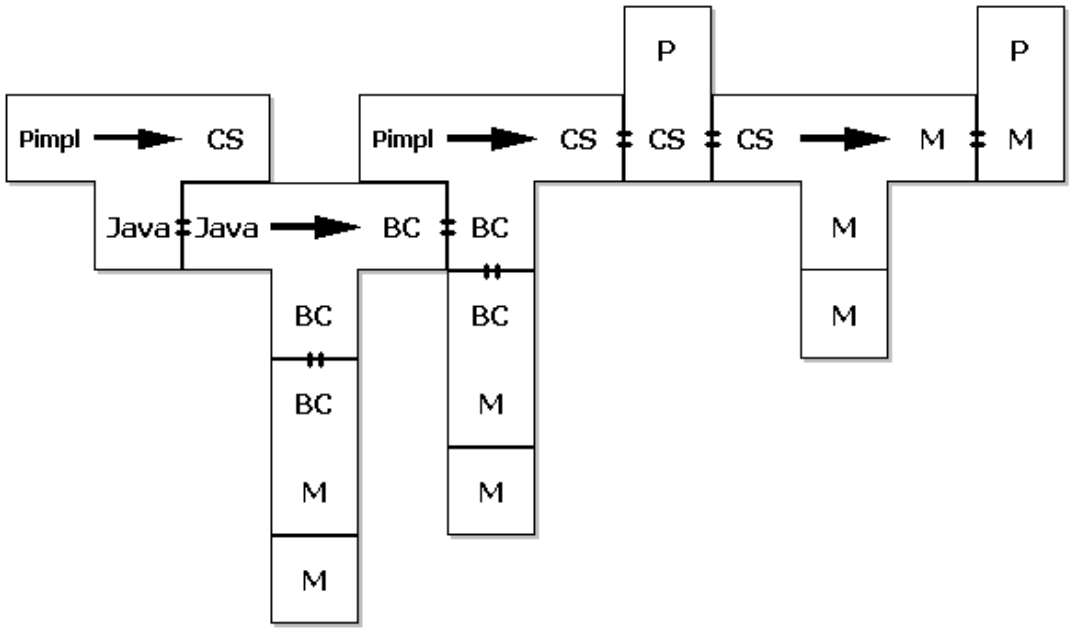


Figure 6.9. Tombstone diagram.

The compiler pipeline is shown in figure 6.9. At first a compiler, that compiles from PIMPL to C#, written in Java, starts the compilation process. We then translate this Java compiler into a compiler written in Java byte code(BC). The compiler written in Java is translated into Java byte code. This gives a compiler, which translate PIMPL program into C# programs. These C# programs are then translated into machine code using the C# compiler from Microsoft.

6.4.1 PIMPL runtime

PIMPL is, as mentioned, translated to C# that uses the .NET framework. This means that all PIMPL programs are running on top of the .NET virtual machine called Common Language Runtime (CLR). All errors not handled by the PIMPL runtime library will cause an error in the CLR.

Runtime library

The library "PimplLib" is the core of the PIMPL runtime environment. A class diagram of "PimplLib" can be seen in appendix H. The "PimplLib" contains a primary class called "Pimpl". All PIMPL programs constructs one instance of this class. The "Pimpl class" includes all the base properties and methods for a PIMPL program e.g. the method "LoadAllFile", that loads all the files in the specified workfolder, inside the program.

The "Pimpl class" implements wrapper methods, to the underlaying Microsoft Windows API calls, for all the file operations used in the language including move, remove, copy, rename and the retrieval of metadata. The wrapper methods all include some error handling and PIMPL specific features not included in the standard libraries, "System.IO" and "Microsoft.WindowsAPICodePack.Shell" containing the API calls. "PimplLib"

contains a class called "File", which is used all the places where files are being used during the execution of a PIMPL program. This class or type is not present in the PIMPL language, but is used to create a good structure in the target code.

The "PimplLib" also includes declarations of some types used in PIMPL e.g. "Time". These types are both new types and types that extends or wraps around existing .NET types.

The reason for creating a library for PIMPL instead of generating all the code at compile time, is to minimizes the duration of the compiliation process and to simplify the generation of code. Almost all the code in the "PimplLib" is used for every program which does not make sense to compile this code every time.

6.4.2 Implicit variables

The "ThisFile" primary expression of Pimpl causes some code generation issues which have been addressed in the compiler. "ThisFile" must be accessible in selector subprograms. "ThisFile" must also be accessable in the "ForeachFile" statements, and as parts of expression in actual parameters of Do-operations. This means that the generated code always must be aware of the file which it is currently working on.

This is in part solved by letting the relevant subprograms, when translated to C#, have an implicit _file argument as their first argument. An example of a translation of a PIMPL Selector, seen in listing 6.14, to a C# method, that includes an implicit _file argument can be seen in listing 6.15.

```
Selector NameSuffix(String s)
{
    If (ThisFile('FileName') == "/.*/" + s)
        Selected;
    Else
        Deselected;
}
```

Listing 6.14. The declaration of the NameSuffix Selector.

```
public static bool NameSuffix(PimplLib.File _file_1, string s)
{
    bool _IsSelected = false;

    if(pimpl.GetMetaData(_file_1, "FileName") == ((new PimplLib.Regex(".*"))
        + s))
        _IsSelected = true;
    else
        _IsSelected = false;
    return _IsSelected;
}
```

Listing 6.15. Translated version of the NameSuffix Selector, with implicit _file_1 argument.

A problem arises with the **ForeachFile** statement, because PIMPL allows nested **ForeachFile** Loops. The implicit variable for the current file must have a different name

for each nesting level because of the scope rules of C#. A variable declared in an inner scope cannot hide a variable with the same name of an outer scope in C#. For this reason, the code generation keeps track of the current implicit file name nesting level using an integer called **implicitFileNestingLevel** and some associated methods called **openImplicitFileNestingLevel** and **closeImplicitFileNestingLevel**.

A method called **getImplicitFileName** is then used whenever the current valid name for the implicit file is needed. This method guarantees to return the correct name for the file reference as long as the methods **openImplicitFileNestingLevel** and **closeImplicitFileNestingLevel** are called whenever "ThisFile" primary expression should have a new meaning.

The name generated by the **getImplicitFileName** method always starts with an underscore "_" symbol. This is done because this guarantees that no name clashing between the implicit variable and user defined variables will occur. This is guaranteed because variable names are not allowed to contain underscores in PIMPL as see in appendix D.

6.4.3 Persistent variables

Persistent variables in PIMPL are like static variables used in subprograms in the C language, where the value of the static variable is retained between subsequent calls to the same subprogram. Only in PIMPL persistent variables are reinitialized just before every rule is executed. C like static variable in subprograms are unfortunately not supported in the target language of the PIMPL compiler C#.

A way to implement similar behavior was needed. Similar behavior, to the C language's static variable in subprograms, can be obtained by utilizing class level static variable in C#. The project group came up with a solution: The code generator saves persistent variable declaration nodes in a list whenever it meets them. It then emits them all, as class level static variables, as the last thing, i.e. when code for all subprograms have been emitted. Name clashing would become a problem if a program, written in PIMPL, was written with two or more subprograms using the same name for a persistent variable. This name clashing issue is solved by adding a prefix to the name of new class level static variables. This prefix contains an underscore, to avoid general name clashing with variables not defined in the PIMPL program, and the full name of the subprogram that declared the persistent variable. Thus the PIMPL Selector declaration of listing 6.16 will be translated to the code in listing 6.17.

```
Selector TestSelector()
{
    Persistent Int testValue = (3 + 3 + 3 + 3) / ((2 + 2 + 2) * (1 + 1));
    testValue = testValue + 1;
    Selected;
}
```

Listing 6.16. Pimpl Test selector declaration.

```
public static bool TestSelector(PimplLib.File _file_1)
{
bool _IsSelected = false;
_persistent_TestSelector_testValue = (_persistent_TestSelector_testValue + 1);
_IsSelected = true;
return _IsSelected;
}
public static int? _persistent_TestSelector_testValue;
private static void _initializePersistentVariables()
{
_persistent_TestSelector_testValue = ((((3 + 3) + 3) + 3) / (((2 + 2) + 2) *
    (1 + 1)));
}
```

Listing 6.17. Translated version of the selector in listing 6.16.

This appended prefix causes rather long variable names, but this is not an issue other than for readability since C# variable names can be arbitrarily long [Microsoft, 2008].

Initialization of persistent variables is done in a special static function called **_initializePersistentVariables**, because the persistent variables will have to be reinitialized before every rule call.

6.4.4 Expressions

A series of parenthesis are emitted, to the C# target code, along with any expression in PIMPL in order to enforce the precedence of operators and the order enforced by parenthesis in expressions in PIMPL. Information about the desired evaluation order will be lost, as it can be seen in listing 6.18, 6.19, and 6.20, in translation if this is not done.

```
(3 + 3 + 3 + 3) / ((2 + 2 + 2) * (1 + 1)) = 1
```

Listing 6.18. PIMPL Expression.

```
3 + 3 + 3 + 3 / 2 + 2 + 2 * 1 + 1 = 15.5
```

Listing 6.19. C# expression without parenthesis.

```
((((3 + 3) + 3) + 3) / (((2 + 2) + 2) * (1 + 1))) = 1
```

Listing 6.20. C# with parenthesis.

6.4.5 Code emission

A PIMPL program requires a certain amount of fixed static code which makes up the PIMPL Runtime environment. Some of this code should be compiled along with the code generated by the PIMPL compiler and some of it is static and can be compiled prior to a library called “PimplLib.dll”.

It is therefore convenient for the compiler to emit rather large blocks of code and to copy already compiled files, in order to make it more standalone. An emit file method has been declared, for the purpose of emitting large static blocks of code. The static code blocks are then saved in files and referenced and emitted during the code generation phase of the PIMPL compiler.

6.5 Error handling

This section describes how PIMPL compilation and runtime errors are handled.

6.5.1 Error handling during compliation

Errors found by a compiler should not prevent it from checking for more errors. The compiler should not stop the contextual analysis of the code if for instance the compiler finds an expression that is not legal, it should merely report an appropriate error message and attempt to continue the contextual analysis. The problem is that a code segment that is considered illegal might be referenced elsewhere, which could result in error messages because the error chains through several code segments, causing otherwise legal code to be erroneous because it is somehow related to a code segment with errors in. The good compiler should only report exactly one error message per unique error.

The PIMPL compiler will, when it finds errors attempt to provide its user with meaningful error messages.

The Pimpl compiler has 2 different categories of error messages during compilation. The first type of error is a scanner or parsing -error. A scanner error occurs if the scanner meets a symbol or some combination of symbols of which it fails to find a matching lexical element in the language, as specified in appendix D. The compiler will in the case of an illegal symbol output a message about the offending symbol and where it is located. This message originates from exceptions thrown by code generated by JavaCC.

A parsing error is found if the input program, in the shape of tokens provided by the scanner, fails to be parsed because it does not conform to the rules of the CFG used to implement the parser. Similarly an error message, which originates from JavaCC generated code, is reported to the user in case of a parsing error. All possible errors in the first category will stop the execution of the compiler and the compiler will only report a single error. This single error policy contradicts with the error handling ideal described earlier but changing this policy has been down prioritized in order to keep the project within its schedule.

The second type of error message originates from the remaining parts of the compiler, the contextual analysis. A new entry is put in the error log if the contextual analysis finds an error. This way the compiler doesn't have to stop once it finds a error. It creates a list of errors and continues and then reports them all once the entire AST is processed.

6.5.2 Runtime errors

There is currently little error handling in the runtime of PIMPL in its current state. Runtime errors related to file operations will be caught in the "PimplLib", and the file operation will simply be skipped. Other runtime errors, i.e. exceptions, would be caught by the Common Language Runtime(CLR) of which the translated C# code is running on but would never be propagated, to the user, in a meaningful way other than a standard error pop-up window provided by the CLR runtime. The program terminates once this CLR runtime pop-up window is closed.

6.6 Testing

As testing is not a primary focus of this project, testing is limited to be done ad-hoc under development of the compiler and for the test cases used on the existing solutions. The testing of the compiler under development and the three test cases are described in this section.

6.6.1 Testing during development

During the design and implementation of the compiler, the different parts of the compiler have been tested. This have mostly been ad-hoc testing and not fully structured testing. This is chosen because ad-hoc testing is less time-consuming than structured testing, but a number of errors is still discovered. A consequence of this shortcut in testing is errors not discovered. Small PIMPL programs, which are syntactically and semantically correct according to PIMPL grammar, were used to test the behavior of the individual phases and the combined phase.

The method has more precisely been to test small parts of the compiler when they were implemented. Some minimal PIMPL programs were written to test the newly implemented parts. At first the only compiler part that was being tested was the syntax analysis part. As more and more parts of the compiler were created, the contextual analysis and later the code generation part were tested. During all the tests, the test programs was modified in a number of ways, to find more errors.

The result of this method of compiler testing lead to a number of errors being located and corrected.

As an example, with the implementation of the Rule-part in the syntax analysis, only a rule part of a PIMPL program was tested, see listing 6.21. If the focus was on the implementation of the Select and the use of Selectors, a new program was written that used a lot of different Selectors with various numbers of parameters. These programs was also modified, as it was tested with both one and more Selectors. Note that the library does not need to be implemented in this test, as it only tests the syntax.

```
Use "MyLibrary.lib";
WorkFolder "C:\Users\InigoMontoya\Pictures\";
RunOption Once;
WorkInSubFolders No;

Rule MyRule1
{
        Select NOT DuplicatesByName() AND Type(".jpeg") OR
            YearBetween(01.01.2010,01.01.2014);
        Do Delete();
}
```

Listing 6.21. Test program with Select.

6.6.2 Full compiler test

PIMPL is tested using small meaningful programs, these programs covers as much of PIMPL's functionality as possible. The cases that the test programs solves are described

in the following section. For each of the solutions for the test cases, the result and possible errors encountered are described.

The purpose of these test were to see how well simple programs could be constructed in PIMPL. The test programs are included in the appendix I.

This section contains an overview of the test cases, followed by an description of the results. The end of the sections includes a summary of the test results.

Test cases

To be able to compare PIMPL with the four existing solutions, PIMPL had to be tested in the same three problems, that is described in appendix A. These three problems are based on the use cases from section 2.1.2. These three problems are listed below.

1. **Organize pictures**: All pictures created in 2013 within a folder must be placed in a new folder named by their creation year "2013". In this newly created folder, subfolders according to each month of the year, should be made, and sorted into these if pictures exist from that month.
2. **Rename pictures**: All pictures within a folder and associated subfolders must have their "Date taken" as a prefix to their names.
3. **Delete duplicates**: Delete all duplicated files located in a specified folder.

Three PIMPL programs are created to test how well the language holds op to the already tested existing solutions.

For the first two problems, seven pictures were used to test these. The pictures had the following 'Photo+DateTaken' properties:

- 30 May 2008
- 23 February 2013
- 8 April 2013
- 27 April 2013
- 2 June 2013
- 12 June 2013
- 17 July 2013

The Pictures used for the third test were mostly copies, but some were not. The names of the files are:

- IMAG0235.jpg
- HEJ - Kopi (2).jpg
- IMAG0235 - Kopi.jpg
- IMAG0235 - Kopi (n).jpg, created ten files, where $2 \leq n \leq 11$.

Problem 1 - Organize Pictures

In the first problem it is necessary to organize files in a folder based structure after year and month e.g. "*\2013\January\".

To facilitate this problem two Calculators and two Selectors were created. At first Selectors are specified to select which files to perform Do-operations on. In listing 6.22 the Selectors are shown.

```
Library:

# Selector which check if a date is between two dates #
Selector DateRange(Date d1, Date d2)
{
        Date pDate = ThisFile('Photo+DateTaken');

        If(pDate >= d1 AND pDate <= d2)
                Selected;
}

# Selector which check FileExtension #
Selector FileType(String s)
{
        If(ThisFile('FileExtension') == s)
                Selected;
}
```

Listing 6.22. Selector library which contains two Selectors.

Listing 6.22 contain two Selectors, "DateRange" and "FileType". "DateRange" takes two parameters of type Date. It then checks if the ThisFile('Photo+DateTaken') value is between the two input parameters, if this is true, then it selects the specific file. "FileType" takes a String as parameter and then check if the ThisFile('FileExtension') value is equal to the input string.

In listing 6.23 parts of the two Calculators can be seen.

```
Library:

# Calculator to check if a file is from 2013 #
Calculator<String> YearCalc(Date d)
{
    If (d >= 01-01-2013 AND d <= 01-01-2014)
        Result = "2013";
    Else
        Result = "Not 2013";
}

# Calculator to calculate which month a file is from #
Calculator<String> MonthCalc(Date t)
{
    If (t >= 01-01-2013 AND t <= 01-02-2013)
        Result = "01-January";
    Else If (t >= 01-02-2013 AND t <= 01-03-2013)
        Result = "02-February";

    #    ...    #

    Else
        Result = "Unknown";
}
```

Listing 6.23. Parts of the Calculator library.

Listing 6.23 contains two Calculators, "YearCalc" and "MonthCalc". The two Calculators does not show the entire code, but only a small code snippet to show a repeating pattern. Both Calculators take one parameter, which is a Date.

The four subprograms created are used in the Rule-part. As seen in listing 6.24, Select contain two subprograms, a "FileType" and "DateRange". "FileType" takes a set as input, and "DateRange" takes two dates. In Do, a Move operation is performed and moves the selected files to the location F:\PIMPL FINAL\Result\Images\2013\January\.

```
# Two set defining a list of strings #
Set<String> ImageTypes(".jpg", ".jpeg", ".png", ".gif", ".bmp", ".JPG");

# Sort images into folders depending on date taken. #
Rule OrganizePictures
{
        Select FileType(ImageTypes) AND DateRange(01-01-2000, 31-12-2015);
        Do Move("F:\PIMPL FINAL\Results\Images\" +
            YearCalc(ThisFile('Photo+DateTaken')) + "\\" +
            MonthCalc(ThisFile('Photo+DateTaken')) + "\\");
}
```

Listing 6.24. Move operation.

The Rule-part shown in listing 6.24 is only a snippet of the complete source code.

The Organize Pictures program was able to solve the problem, however the solution proved to not be as elegant as the group had hoped. The solution for the problem required around 80 lines of code. The full code can be seen in appendix I.

Problem 2 - Rename Pictures

The second problem is about renaming files. The files being renamed are pictures and have to be renamed to the value of the 'Photo+DateTaken' metadate, followed by the original 'FileName'. The program needs files of the relevant types, and used the "FileType" Selector, from problem 1.

In the Rule part, as seen on listing 6.25, a Rename operation is defined in the Do operation. It renames the selected files with the 'Photo+DateTaken' metadata and original 'FileName', while a dash(-) separate these two.

```
Rule OrganizePictures
{
        Select FileType(ImageTypes);
        Do Rename(ThisFile('Photo+DateTaken') + " - " + ThisFile('FileName'));
}
```

Listing 6.25. Rule showing Rename.

Renaming pictures with the prefix of 'Photo+DateTaken', followed by the original 'FileName', does not require a large amount of code to accomplish. A solution for this problem took up 23 lines of code in PIMPL. The full code can be seen in appendix I.

Problem 3 - Delete Duplicates

In PIMPL deleting duplicates is done by using a hash value of files and compare these to select duplicates. This Selector is shown in listing 6.26. This Selector makes sure that the file found is not itself. Using the ForeachFile, it checks all files according to the conditions and specify if it should be selected or deselected.

```
Library:

Selector Duplicates()
{
        String temp = ThisFile.HashValue;
        Int tempInt = 0;

        ForeachFile
        {
                If(ThisFile.HashValue == temp AND ThisFile.IsSelected ==
                    False)
                {
                        tempInt = tempInt + 1;
                }

                If(tempInt > 1)
                {
                        Selected;
                }
        }
}
```

Listing 6.26. Selector to choose duplicates.

The Rule-part for deleting duplicates consist of a single Rule, containing a Selector in Select and a delete operation in the Do declaration. The Delete() does not take any arguments, but deletes the selected files. The code snippet for the Rule-part can be seen in listing 6.27.

```
Rule DeleteDuplicates
{
    Select Duplicates();
    Do Delete();
}
```

Listing 6.27. Rule for DeleteDuplicates.

A PIMPL solution for this problem took up 30 lines of code. The full code can be seen in appendix I.

Summary of testing

The three test cases, leads to a few conclusions. Firstly it was possible to create a working solution for all three problems. Organizing pictures showed that a way to manipulate the Date type, and possibly other types, could be a beneficial addition to PIMPL. For the current solution, a number of things have to be hard coded.

6.6.3 Extended testing

PIMPL should ideally be tested in multiple different ways, if a public release should happen. Structured testing of all the functionality available in PIMPL should be done, in order to ensure that the functions work as intended. Ideally, user tests with the target group are needed to ensure that the language have the features the target group could use. The compiler needs to have a more in depth quality test. There could be made unit testing, to ensure that the different parts of the compiler works as intended.

7 Discussion

The following discussion is included here in order to evaluate the choices made during the project. The chapter presents a discussion concerning the developed programming language PIMPL. This discussion concerns two primary subjects: the evaluation of the programming language and the project approach followed. The evaluation of PIMPL covers a comparison between PIMPL and existing file managing software, a discussion on the limitations and shortcomings, and an evaluation of the paradigm choice and the language evaluation criteria. The method section covers the evaluation and discussion of the used development model, the used JavaCC and JJTree tools for creating scanner/parser, the consequences of not having a group of users and the evaluation of the semantics.

7.1 PIMPL language evaluation

This section includes the project groups evaluation and discussion related to the developed PIMPL language.

7.1.1 PIMPL in relation to the existing solutions

The project group noted the following, for the examined existing solutions: The languages contained all the functionality, the group found relevant, when performing file organization tasks, although the languages are cumbersome to use when writing file organizing programs e.g if the user want to load some metadata. The programs tested proved to be user-friendly and intuitive to use, but they lacked certain functionality concerning the use of metadata and advanced control structures.

The project group finds that the PIMPL language contains an understandable, user-friendly syntax for the target group. PIMPL contains functionality that allows users to use metadata when organizing files. PIMPL is the only solution, compared to the other languages, that has build-in functionality that allows the use of metadata.

Case	PIMPL	Java	C# (Target code)
Organize Pictures	81 lines	129 lines	195 lines
Rename Pictures	23 lines	85 lines	119 lines
Delete Duplicates	30 lines	N/A	147 lines

Table 7.1. The number of lines required to solve the tasks.

As seen in table 7.1, the amount of code required to solve the same task in the different

languages, show that PIMPL uses considerably less code, as compared to Java and C#. This could lead to some degree of increased writability, from the perspective of the target group. The project group assumes that the writability may not be better for more experienced users who are trying to learn PIMPL.

Some of the structures in PIMPL are highly abstract, which means the users does not need to worry about e.g. the file related problems. An example of a file related problem is when a user tries to delete a non-existing file. These kind of errors must be dealt with by the users in other languages like Java or C#, but in PIMPL the Delete() command always performs runtime checks to ensure that the action is legal. Overall this means that less work is required for the users to code file organization programs in PIMPL.

7.1.2 PIMPL limitations

This section addresses some of the shortcomings the PIMPL language contains.

An idea to a feature that seems useful could be the opportunity for users to create their own Do-operations, similar to the Selectors in the language. It seems weird that there exist so many different ways to define which file should be chosen exactly, when only a fixed number of Do-operations exist. Although only a limited number of file operations exist. A possibility to group operations could also be useful.

The use of additional work folders, if work folders could be declared and then for each Rule set the folder which that particular rule should operate at. At the moment, this is not possible. The users must write a new program if additional work folders should be used. Copy/paste might be used to just copy Rules into new programs, where the only difference is the defined WorkFolder.

PIMPL includes implicit casting (Coercion) which allows conversion between compatible types, but in a few cases an explicit cast could be a useful feature e.g. converting a String to an Int. Explicit casting (Conversion) would be a feature targeted experienced users, whom develops libraries in PIMPL.

PIMPL supports regular expressions but only for pattern matching. It could be useful if the regular expressions were able to extract a part of a variable e.g. if the user want only to look at only the month in a Date variable.

It has been difficult to design the runtime error notifications of PIMPL. The group has discussed the extent of the notifications that the users are exposed to. PIMPL is meant to run in the background and not disturb the user unnecessarily. It is sometimes not necessary to disturb the user e.g. if the program tries to delete a non-existing file. The program then just skips the file and continues with the remaining files in the WorkFolder. It is, in other cases, necessary to disturb the user e.g. if the WorkFolder does not exist, then it makes no sense to continue executing and the PIMPL program throws an error. The project has had a lack of user interaction resulting in difficulties answering whether or not it is a good solution.

It could have been useful if the PIMPL language contained more symbol tables. This way, it could be possible to make variables, Rules, Selectors and Calculators with the same

name.

7.1.3 Paradigm choice

This section addresses the evaluation of the choice of the imperative programming paradigm.

The imperative programming paradigm was followed when designing the PIMPL language. The imperative paradigm was used in order to handle the Rules, as to get an aspect of sequence. The Rules specified might first complete a task that copy some files from a folder A to another folder B, and the next rule could delete all files in folder A. The result differs, depending on what rule runs first.

Designing with the chosen paradigm in mind gave PIMPL sequence, that provided a good way to execute the rules in a program. The imperative paradigm was satisfying to model after, although a declarative paradigm was wanted at first, but as complications made this unable for us, the imperative paradigm was followed.

7.1.4 Language evaluation criteria

This section includes an evaluation of the language evaluation criteria. The criteria themselves are evaluated, to see if these were prioritized correctly and if some criteria should have been different. Following this is an evaluation of PIMPL, where PIMPL is evaluated for each of the criteria.

Evaluation of criteria

This section evaluates at the criteria chosen for the project. This is in term of their prioritizing. This section is only concerned with the faults and choices that should have been different.

The project group has considered if the orthogonality criteria should have been prioritized higher in the design phase, as a high level of orthogonality makes a programming language easier to learn, read and write. These gains are substantial, and gives a more uniform language. As the goal was to make an easy readable, learnable language, orthogonality should have been considered more when designing the language.

Another thing is the reliability criteria, the group thought that this maybe should have been prioritized higher, as the safety of the users files and the assurance that it is the correct files being organized is very important. If it is possible for anything unexpected to happen to important files, the reliability for the language decreases.

Some criteria have not been given a high priority, because the language and the accompanying language processor was never intended for release. The maintainability-, efficiency- and portability criteria, should have gotten higher priority, if the language was to be released.

Evaluation of PIMPL

The criteria are evaluated by the project group and not users, because the project lacks user interaction. The criteria are evaluated in the order mentioned in the report. The criteria order and priority can be seen in table 3.1.

The project group thinks that the readability of the language is very good. The keywords chosen and syntactic elements used makes a language that seems easily manageable. The keywords must be written in Pascal Case, this makes the keywords stand out and the keywords used, clearly states the meaning of the construct.

The writability criterion was prioritized as Important, and this criteria has partially been fulfilled. We wanted a language that has a high level of writability, but the project group did not want to compromise the readability criterion, and so readability was prioritized highest. Though language constructs that supports writability exists in the form of "sets". Another read- and writability improvement, that the project group discussed, was the ability to write Yes and No, along with True and False, for boolean values. Novice users might find it easier to understand a boolean value if it could be written Yes and No. If PIMPL users decide to learn a new programming language, they might have formed a habit of writing Yes and No, instead of using True and False, which is the convention in C like programming languages.

Next is reliability that too were set at Important. For the reliability part, we included a type checker into the compiler, which contributes to the reliability. When manipulating files automatically, an amount of safety is needed, as important files might get deleted. We made sure that files only were moved to the recycle bin, instead of being permanently deleted.

A user manual is provided for the language, intended to help the users with writing programs in the PIMPL language. This user manual can be seen in appendix I. The language constructs are simple and the language contains an intuitive segregation of the Options stated first, followed by the Rules.

Implementability was set at Very important. This was because a goal of the project to implement a working language processor for the created programming language.

Orthogonality was marked as Less important for the project. This is because the project group thinks that orthogonality probably makes it easier to learn if things can be combined in many different ways. The language ended being very orthogonal and the language structures can not be combined in many different ways.

The efficiency criterion was set at Less important. The running time of a PIMPL program is relatively fast. The bottleneck of PIMPL programs are the file operations, and these depends on the size and amount of files. The project group finds that the file operations performs as fast as when using the Windows GUI for file operations.

The remaining criteria were prioritized as Not important, and these were not further taken into consideration when developing PIMPL. Although these criteria should be considered if PIMPL was to be released.

7.2 Method evaluation

This section includes the project groups evaluation and discussion related to the method part of the project.

Development approach evaluation

The project group used a combination of two development approaches, the waterfall- and iterative model. The main approach has been the waterfall model. This approach was used on everything but the design and implementation of the language. The design and implementation of the language has been an iterative process. The need for several iterations for this has been necessary because it is a new domain for the project group. The language design and implementation was gradually improved as the project group gained knowledge on the subject from semester courses. Overall this development approach for the project has been good. Only using the waterfall approach, could have saved time, but as new knowledge was gained, the iterative approach was needed.

Compiler compiler tools evaluation

The project group decided to use tools to help generate code for the syntactic analysis part of the compiler. Given the project group's experience in compiler creation, a handwritten compiler could potentially introduce errors that might not have occurred with a well established tool.

There exist a number of different compiler compiler tools. It is likely that another tool could have been better suited for the project. JavaCC did not have easily obtainable documentation, when more advanced features were needed.

LL(1) seemed like the most obvious choice, but it may not have been the correct choice to make. This lead to some difficulties, like grammar- and AST transformation.

Lack of user interaction

The project is build on the basis of the project group members own experience, opinions, thoughts and ideas and some few external Internet sources. The project does not include any user interaction. It has been difficult to create a language and take some decisions about the language to a target group without having any interaction. The project group believe the language is useful for the target group because the overall syntax for the language is very simple compared with other languages and the user would work on a high level of abstraction.

Semantics evaluation

The project do not contain a full formal semantics because of the time limitations. A full formal semantics would ensure that all parts of the language were well defined.

Conclusion 8

Based on a discussion in the project group, the group selected file organization as the initial problem. This project then continued with short analysis of the initial problem and a possible target group for a new file management programming language. A problem definition for a new programming language was formulated on the basis of this analysis. It was found that the new programming language should be a domain specific language with focus on organization of files.

The design of the language started out with the creation of a set of language evaluation criteria. The project group chose that the new programming language should be highly abstract, within the domain of the language and that it should be imperative. These criteria served as a basis for the design of a programming language which was named PIMPL.

The project group chose to implement a compiler, as opposed to an interpreter, to implement the new programming language. From these choices, the PIMPL programming language was designed. This consist of a formal definition of the syntax, which includes a definition of lexical elements and a context free grammar. Semantics for a subset of the language was defined both formally and informally.

The project group has designed a programming language which takes programming of file organization with file metadata to a higher abstraction level than Java. A compiler for the PIMPL language was implemented, which allows the translation of PIMPL programs. This enables a user to automate file organization in the Windows operating system using PIMPL.

Several tests of the compiler, on programs based on a series of language test cases, was conducted during the implementation and after the implementation. These tests helped locate compiler errors during implementation. After implementation, some tests proved that programs performs as expected.

The project group has designed and implemented a language that, in our opinion, makes it easier to develop programs, for automating file organizing tasks in the Windows operating system. The project has resulted in a language that, in our opinion, meets the expectations of the potential target group.

Further Development 9

The project group would like to add a number of additions and changes, to the the PIMPL language and compiler, if the development was to continue. It would be ideal to include a test group, consisting of the primary and secondary target group, if the development was to continue. These could be interviewed to give a foundation for discovering possible problems and the possible new features to the language. It would be ideal if users could test the PIMPL language. The project group think that a set amount of future features would help improve PIMPL and how it is used. The project group finds that more in depth options for file operations could improve the experience of PIMPL. A list of these features is shown below.

- Define multiple work folders
- Do expansion
- Split Date and Time
- Folder operations
- Type casting
- Improved Regular Expressions
- Collection of variables

Define multiple work folders
It could be useful to be able to define multiple work folders. This could make it possible to work in folders at different locations. An addition to this could be be able to define a work folder in a rule, so a specific rule runs on the defined work folder for that rule.

Do expansion
In a Library, the user can specify its own Selectors. It could open up to a wide variety of opportunities for the user, if it was possible for the user to define a Do-operation in a similar way, as the Selectors. A limited number of file operations exists, but the possibility to create combinations of these could be useful.

Split Date and Time
Being able to take a Date and Time apart and only read a specific month or year, could be useful. It would help when extracting a specific part of a metadata value in a Date and Time. This could be useful when the goal is to sort image files based on the year and month the images were captured.

Folder operations
It is possible to move all files in a folder to a new folder, but moving a folder to a new destination is not possible. It would be useful to be able to copy a folder to a new destination before performing other operations on that specific folder.

Type casting
Being able to explicitly cast a type to another type would be fairly useful, e.g. a String value could be converted into an Int value, for the purpose of comparing it to another Int value.

Improved Regular Expressions
Be able to use regular expression to extract a part of a value e.g. if the user want to use only the minutes in a Time variable.

Collection of variables
It would be an improvement to be able to create lists, where it is possible to add and remove variables. This could be used in Selectors and Calculators.

There seems to be a number of future language features to add to PIMPL, but the focus should still be to keep PIMPL a domain specific language and not turn it into a general purpose language. The mentioned possible future additions listed, could be expanded if users were included in the development process. This could help improve PIMPL and increase organization possibilities and customization of how users wish to organize files.

Bibliography

Alverno College. What are file operations, 2012. URL http://depts.alverno.edu/cil/basic/filemanage/operations.html. Last seen March the 12th, 2014.

ANTLR. ANTLR, 2014. URL http://www.antlr.org/.

Antoon Bosselaers. The hash function RIPEMD-160, 2012. URL http://homes.esat.kuleuven.be/~bosselae/ripemd160.html.

Deborah Barreau and Bonnie A. Nardi. Finding and Reminding, 1995. URL http://dl.acm.org/citation.cfm?id=221307. Last seen March the 5th, 2014.

c4learn. Difference between Compiler and Interpreter, 2014. URL http://www.c4learn.com/c-programming/compiler-vs-interpreter/.

Cem Ozdogan. File Operations, 2011. URL http://siber.cankaya.edu.tr/OperatingSystems/ceng328/node210.html. Last seen March the 5th, 2014.

Charles N. Fischer, Ron K. Cytron, and Jr. Richard J. LeBlanc. *Crafting a Compiler*. Pearson Education, 2009. ISBN 978-0-13-606705-4.

Hanspeter Mössenböck and Markus Löberbauer and Albrecht Wöß, University of Linz. The Compiler Generator Coco/R, 2011. URL http://www.ssw.uni-linz.ac.at/Coco/.

Hans Hüttel. *Transitions and Trees: An Introduction to Structural Operational Semantics*. Cambridge University Press, 1 edition, 2010. ISBN 978-0-521-14709-5.

Instagram. Instagram statistics, 2014. URL http://instagram.com/press/#. Last seen April the 15th, 2014.

Java. About Java, 2014. URL http://java.com/en/about/. Last seen March the 5th, 2014.

Java. Learn About Java Technology, 2014. URL https://www.java.com/en/about/. Last seen March the 12th, 2014.

JJTree. JavaCC [tm]: JJTree Reference Documentation, 2014. URL https://javacc.java.net/doc/JJTree.html.

Lupo PenSuite Team. DropIt project, 2014. URL http://www.dropitproject.com/. Last seen February the 21th, 2014.

Marjan Mernik and Jan Heering and Anthony M. Sloane. When and How to Develop Domain Specific Languages. *ACM Computing Surveys*, 12 2005. http://dl.acm.org/citation.cfm?id=1118892.

Microsoft. Maximum length of identifer i visual studio C#, 2008. URL http://msdn.microsoft.com/en-us/library/t94za9fd(v=vs.90).aspx.

Microsoft. Arbejde med filer og mapper, 2014a. URL http://windows.microsoft.com/da-dk/windows/working-with-files-folders#1TC=windows-7. Last seen March the 6th, 2014.

Microsoft. Arbejde med biblioteker, 2014b. URL http://windows.microsoft.com/da-dk/windows7/working-with-libraries. Last seen March the 7th, 2014.

Microsoft. Regex Class, 2014c. URL http://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex.aspx.

Microsoft. Windows PowerShell Overview, 2014d. URL http://technet.microsoft.com/en-us/library/cc732114(v=ws.10).aspx. Last seen February the 25th, 2014.

Microsoft. Using the clear-content cmdlet, 2014e. URL http://technet.microsoft.com/en-us/library/ee156808.aspx. Last seen March the 12th, 2014.

Microsoft MSDN. Int32.MaxValue Field, 2014a. URL http://msdn.microsoft.com/en-us/library/system.int32.maxvalue.aspx.

Microsoft MSDN. Int32.MinValue Field, 2014b. URL http://msdn.microsoft.com/en-us/library/system.int32.minvalue(v=vs.110).aspx.

Mikey Campbell. NPD: Chromebook sales outperform MacBooks in commercial sector as iPad loses ground. *AppleInsider*, 12 2013. http://appleinsider.com/articles/13/12/29/npd-chromebook-sales-outperform-macbooks-in-commercial-sector-as-ipad-loses-ground.

Kurt Nørmark. Overview of the four main programming paradigms, 2013. URL http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html. Last seen February the 20th, 2014.

Oracle. Lesson: Regular Expressions, 2014. URL http://docs.oracle.com/javase/tutorial/essential/regex/.

Oracle. Java™ programming language, 2014. URL http://http://docs.oracle.com/javase/7/docs/technotes/guides/language/. Last seen February the 26th, 2014.

Adam Pash. Belvedere Automates Your Self-Cleaning PC, 2008. URL http://lifehacker.com/341950/belvedere-automates-your-self+cleaning-pc. Last seen February the 21th, 2014.

Patrick Darling. Stressed by Technology? You Are Not Alone. *Intel Newsroom*, 8 2010. http://newsroom.intel.com/community/intel_newsroom/blog/2010/08/19/stressed-by-technology-you-are-not-alone?cid=rss-90004-c1-258170.

M Angela Sasse Richard Boardman. "Stuff Goes into the Computer and Doesn't Come Out" A Cross-tool Study of Personal Information Management, 2004. URL http://dl.acm.org/citation.cfm?id=985766. Last seen March the 5th, 2014.

Robert W. Sebesta. *Concepts of Programming Languages*. Pearson Education Limited, 10 edition, 2013. ISBN 978-0-13-139531-2.

Stephen Shankland. Dropbox clears 1 billion file uploads per day, 2013. URL http://reviews.cnet.com/8301-13970_7-57571513-78/dropbox-clears-1-billion-file-uploads-per-day/. Last seen March the 5th, 2014.

Danmarks statistik. Sortere og gemme filer på computeren så de nemt kan findes igen, 2009. URL http://www.statistikbanken.dk/statbank5a/SelectVarVal/Define.asp?MainTable=ITF&PLanguage=0&PXSId=0&wsid=cftree. Last seen March the 5th, 2014.

Stefan Brunthaler. Why Interpreters Matter, 2012. URL http://www.ics.uci.edu/~sbruntha/why-interpreters-matter.html#hsinterp.

Stock Artists Alliance. META Resources - Standards: Exif, 2014. URL http://www.photometadata.org/meta-resources-metadata-types-standards-exif. Last seen March the 5th, 2014.

Teach-ict. Comparing Compiler & Interpreter, 2014. URL http://www.teach-ict.com/as_as_computing/ocr/H447/F453/3_3_2/translators_compilers/miniweb/pg14.htm.

Tiobe. Most Popular Programming Languages, 2014. URL http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Last seen March the 6th, 2014.

W3schools. OS Platform Statistics, 2014. URL http://www.w3schools.com/browsers/browsers_os.asp.

Wikipedia. Design rule for Camera File System, 2014a. URL http://en.wikipedia.org/wiki/Design_rule_for_Camera_File_system. Last seen March the 12th, 2014.

Wikipedia. Functional programming, 2014b. URL http://en.wikipedia.org/wiki/Functional_programming. Last seen March the 4th, 2014.

Wikipedia. Imperative Programming, 2014c. URL http://en.wikipedia.org/wiki/Imperative_programming. Last seen March the 4th, 2014.

Wikipedia. Information Management, 2014d. URL http://da.wikipedia.org/wiki/Information_management. Last seen April the 14th, 2014.

Wikipedia. Functional programming, 2014e. URL http://en.wikipedia.org/wiki/Logic_programming. Last seen March the 4th, 2014.

Wikipedia. Object-oriented programming, 2014f. URL http://en.wikipedia.org/wiki/Object-oriented_programming. Last seen March the 5th, 2014.

Wikipedia. JPEG - Typical usage, 2014g. URL http://en.wikipedia.org/wiki/JPEG. Last seen March the 5th, 2014.

Harry Bruce William Jones, Abe Wenning. How Do People Re-find Files, Emails and Web Pages?, 2014. URL https://www.ideals.illinois.edu/bitstream/handle/2142/47300/136_ready.pdf?sequence=2. Last seen March the 5th, 2014.

Existing solution test A

This appendix presents the problems stated in order to test a solution, whether it being a program or a programming language. Afterward the results from each existing solution is presented. This appendix is associated with section 2.3.

Problems and prerequisites

The problems are described in this section. When writing and testing the programs and languages, we found out that the solutions only can access the file metadata, which is the same for every file type focused in this project. Therefore it does not matter what filetype the test is about.

The problems for the test are listed below:

1. **Organize pictures**: All pictures created in 2013 within a folder must be placed in a new folder named by their creation year "2013". In this newly created folder, subfolders according to each month of the year, should be made, and sorted into these if pictures exist from that month.
2. **Rename pictures**: All pictures within a folder and associated subfolders must have their "Date taken" as a prefix to their names.
3. **Delete duplicates**: Delete all duplicated files located in a specified folder.

For the first two problems, seven pictures were used to test these. The pictures had the following "date taken" properties:

- 30 May 2008
- 23 February 2013
- 8 April 2013
- 27 April 2013
- 2 June 2013
- 12 June 2013
- 17 July 2013

The Pictures used for the third test were mostly copies, but some were not. The names of the files are:

- IMAG0235.jpg
- HEJ - Kopi (2).jpg
- IMAG0235 - Kopi.jpg
- IMAG0235 - Kopi (n).jpg, created ten files, where $2 \leq n \leq 11$.

Results

This section presents the results acquired in the tests.

Belvedere

Problem 1
Belvedere does not give access to changing files which has been taken inside a certain date or defined time period. But the program can access files which has been created inside a given number of e.g. weeks ago, but with the current date as the high boundary. It is therefore not possible to move files taken a certain year and moving these to a folder showing this. But it was possible to move all files from the last 70 weeks to a folder named "2013".

Problem 2
Belvedere gives the opportunity to create rules for certain file types, which makes it possible to make rules for e.g. pictures or music. The program can change the files, but only to names specified by the program, e.g. renaming files can only be to names indicating the current date and time. If the user chooses to rename files for the current time, the program will constantly rename files in an infinite loop.

Problem 3
Belvedere can easily delete certain files, either by moving these to the recycle bin or by deleting these permanently. Belvedere can have rules that can delete files where the name contains e.g. "- Kopi", Though it is not possible only to delete the copy, if there exist an original version of the same image in the same folder.

DropIt

Problem 1
DropIt cannot only make a folder for the 2013 pictures. No matter what, the program will also make a folder for the 2008 picture, even though only pictures from 2013 has been specified. There are extra filters to sort by, but these does not work. The month folders is regulated by the language the user uses in windows and cannot be specified differently. Although there is some problems and only simplified sorting works, the program still presents a partially accepted result, with fairly small amount of work put in and only few tries were needed.

Problem 2
When all the pictures are placed in the root folder, and the only task is to rename files, is it fairly easy. But putting tasks together, for this problem, renaming and moving files cannot be done. But making two different sorting algorithms and running these one at the time can make it work. Does also work on files in subfolders, and this cannot be avoided, if this is wanted.

Problem 3
It is not possible to remove duplicates in folders. The only thing the user can do, is to specify some part of a name e.g. "- Kopi" to remove all files with these characters in their name.

Java

For the Java solution code, see appendix I.

Common for the Java solutions
It is necessary to set up a few things before being able to create a Java solution. This section describes the prerequisites for creating a Java solution for the problems. Both solutions requires import of a library to handle the reading of EXIF data. A library called "metadata-extractor" is used for both implementations. This required 5 lines of code in order to import the necessary class definitions and the import of two ".jar" files into the project. It would of course be possible to implement a custom EXIF data extractor solution to the problem, but that would be out of the scope of this test since a new language, with files as its primary problem domain, would provide similar easy access to EXIF-reading functionality.

The code includes several exception handling code-lines because Java requires that all so called "Checked Exceptions" must be handled in the program code. The EXIF library used here throws exceptions for instance if the file does not include any EXIF data or if the file does not exist.

Both solutions required specification of an algorithm to traverse all files in the target folder. All of these prerequisites takes up many lines of code.

Problem 1
This solution uses a simple algorithm which traverses all the files in a folder and uses 6 lines not including the actions performed for each file the algorithms find. In total, a Java solution for this problem is 129 lines.

Problem 2
This solutions uses a recursive method called "TraversePathAndDo" this method takes up 14 lines in the code. In total, a Java solution for this problem is 85 lines.

Problem 3
Problem 3 was not made, as the prior problems showed how much code was needed to solve relatively simple file organization problems.

PowerShell

PowerShell offer good possibilities at making operations on files. Meaning that operations such as deleting, rename or moving files based on conditions set is relatively simple. It does get very complicated at accessing file specific metadata. The program must then include a lot of code, which requires time to learn and set up. After using a few hours on this problem we concluded that this was too complicated and can easily be simpler. After this conclusion the project group chose to not solve all the test problems using PowerShell.

Readable EBNF Grammar B

This appendix presents the grammar for the created programming language PIMPL. All the tokens is defined in Appendix D.

```
(* Start *)
<Start>                     = (<LibraryPart> | <RulesPart>) $

(* Rule-part *)
<RulesPart>                 = <Options> <RuleDcls>

<Options>                   = <IncludeLibraryStmt> <WorkFolderDef>
    <SetRepeatOptDef> <WorkInSubOptDef> [<Exclude>] [<RunRules>]
    [<FinallyOnChange>] [<SetDcls>]

(* Options *)
<IncludeLibraryStmt>        = use stringConst { comma stringConst } semiColon

<WorkFolderDef>             = setWorkFolder pathConst  semiColon

<SetRepeatOptDef>           = setRepeatOption ( once | recurring ) semiColon

<WorkInSubOptDef>           = workInSubFolders boolConst semiColon

<FinallyOnChange>           = finallyOnChange [<FinallyOnChangeActions>]
    semiColon

<FinallyOnChangeActions>    = message lparen stringConst  rparen | beep
    lparen  rparen

<Exclude>                   = exclude <Paths> semiColon

<Paths>                     = pathConst {comma pathConst}


(* Set *)
<SetDcls>                   = <SetDcl> {<SetDcls>}

<SetDcl>                    = set langlebracket <TypeName> ranglebracket ID
    lparen <Values> rparen semiColon

<TypeName>                  = intType
                            | timeType
                            | dateType
                            | pathType
                            | stringType

(* Values *)
<Values>                    = [<Value>] { comma <Values> }

<Value>                     = stringConst
                            | pathConst
                            | ID
                            | boolConst
                            | intConst
                            | (timeConst | dateConst)
```

```
(* RunRules *)
<RunRules>                  = runRules <RulesList> semiColon

<RulesList>                 = ID {comma ID}

(* Rules *)
<RuleDcls>                  = <RuleDcl> { <RuleDcls> }

<RuleDcl>                   = rule ID lbrace <SelectProduction> <DoProduction>
    rbrace

(* Select *)
<SelectProduction>          = select <Selects> semiColon

<Selects>                   = <Select> { (logicalOrOperator |
    logicalAndOperator) <Selects> }

<Select>                    = [boolNegation] ID lparen <Values> rparen

(* Do *)
<DoProduction>              = doSymbol <Dos> semiColon

<Dos>                       = <Do> { comma <Dos> }

<Do>                        = <DoOperationSpecifier> lparen <Expression> rparen

<DoOperationSpecifier>      = move
                            | rename
                            | delete
                            | copy

(* Library-part *)
<LibraryPart>               = libraryColon <LibraryDcls>

<LibraryDcls>               = {<SelectorDcl> | <CalculatorDcl>}+

(* Selector *)
<SelectorDcl>               = selector ID lparen <FormalParameterList> rparen
    lbrace <Stmts> rbrace

(* Calculator *)
<CalculatorDcl>             = calculator langlebracket <TypeName>
    ranglebracket ID lparen [<FormalParameterList>] rparen lbrace <Stmts>
    rbrace

<CalculatorCall>            = ID lparen [<ActualParameterList>] rparen

(* Parameter list *)
<ActualParameterList>       = <Expression> {comma <Expression>}

<FormalParameterList>       = <TypeName> ID {comma <TypeName> ID}

(* Statements *)
<Stmts>                     = {<Stmt>}

<Stmt>                      = <VariableDcl> semiColon
                            | <SelectedStmt> semiColon
                            | <ForEachFileStmt>
                            | <IfStmt>
                            | <AssignmentStmt> semiColon

<VariableDcl>               = [persistent] <TypeName> ID [assignment
    <Expression>]

<SelectedStmt>              = selected | deSelected

<ForEachFileStmt>           = forEachFile <BlockOrStatement>
<IfStmt>                    = ifSymbol lparen <Expression> rparen
    <BlockOrStatement> [ elseSymbol <BlockOrStatement> ]
```

```
<BlockOrStatement>          = lbrace <Stmts> rbrace
                            | <Stmt>

<AssignmentStmt>            = ID assignmentOperator <Expression>

(* Expression *)
<Expression>                = <LogicalOrExpression>
<LogicalOrExpression>       = <LogicalAndExpression>
                            | <LogicalOrExpression> logicalOrOperator
                                <LogicalAndExpression>

<LogicalAndExpression>      = <EqualityExpression>
                            | <LogicalAndExpression> logicalAndOperator
                                <EqualityExpression>

<EqualityExpression>        = <RelationalExpression>
                            | equality_expression <EqualityOperator>
                                <RelationalExpression>

<EqualityOperator>          = equalityOperator
                            | notEqualOperator

<RelationalExpression>      = <AdditiveExpression>
                            | <RelationalExpression> <RelationalOperator>
                                <AdditiveExpression>

<RelationalOperator>        = langlebracket
                            | ranglebracket
                            | lesserEqualThanOperator
                            | biggerEqualThanOperator

<AdditiveExpression>        = <MultiplicativeExpression>
                            | <AdditiveExpression> <AdditiveOperator>
                                <MultiplicativeExpression>

<AdditiveOperator>          = minusOperator
                            | plusOperator

<MultiplicativeExpression> = <UnaryExpression>
                            | <MultiplicativeExpression>
                                <MultiplicativeOperator> <UnaryExpression>

<MultiplicativeOperator>    = multiplicationOperator
                            | divisionOperator
                            | moduloOperator

<UnaryExpression>           = <PrimaryExpression>
                            | <UnaryOperator> <UnaryExpression>

<UnaryOperator>             = boolNegation
                            | minusOperator

<PrimaryExpression>         = <ExpressionValue>
                            | lparen <Expression> rparen
                            | <CalculatorCall>
                            | <ThisFile>
                            | regEx

<ExpressionValue>           = stringConst
                            | boolConst
                            | intConst
                            | <Time>
                            | ID
                            | pathConst

<ThisFile>                  = thisFile ( lparen metaDataSpecifier rparen | dot
    isSelected | dot hashValue )
```

Listing B.1. Grammar for PIMPL.

Implementation LL(1) Grammar

C

This appendix show the LL(1) grammar for PIMPL.

```
(* Start *)
<Start>                     = <StartB> $

<StartB>                    = <LibraryPart> <StartC>  <RulesPart>

<StartC>                    = <LibraryPart> <StartC>
                            | EPSILON

(* RULE *)
<RulesPart>                 = <Definitions> <RulesPartB>

<RulesPartB>                = <RunRules> <RulesPartC>
                            | <RulesPartC>

<RulesPartC>                = <FinallyOnChange> <RulesPartD>
                            | <RulesPartD>

<RulesPartD>                = <SetDcls> <RuleDcls>
                            | <RuleDcls>

(* Definitions *)
<Definitions>               = <WorkFolderDef> <SetRepeatOptDef>
    <WorkInSubOptDef> <DefinitionsB>

<DefinitionsB>              = <Exclude>
                            | EPSILON

<WorkFolderDef>             = setWorkFolder pathConst semiColon

<SetRepeatOptDef>           = setRepeatOption <RepeatOption> semiColon

<RepeatOption>              = once
                            | recurring

<WorkInSubOptDef>           = workInSubFolders <BooleanConst> semiColon

<FinallyOnChange>           = finallyOnChange <FinallyOnChangeActions>
    semiColon

<FinallyOnChangeActions>    = message lparen stringConst rparen
                            | beep lparen rparen

<Exclude>                   = exclude <Paths> semiColon

(* Set *)
<SetDcls>                   = <SetDcl> <SetDclsB>

<SetDclsB>                  = <SetDcl> <SetDclsB>
                            | EPSILON

<SetDcl>                    = set langlebracket <TypeName> ranglebracket ID
```

```
    lparen <Values> rparen semiColon

<TypeName>                  = stringType
                            | intType
                            | boolType
                            | timeType
                            | dateType
                            | pathType
                            | regexType

(* Values *)
<Values>                    = <Value> <ValuesB>
                            | EPSILON

<ValuesB>                   = comma <Value> <ValuesB>
                            | EPSILON

<Value>                     = pathConst
                            | ID
                            | <BoolenaConst>
                            | intConst
                            | <Time>
                            | stringConst
                            | regexConst

<BooleanConst>                    = boolConst

<Time>                      = timeConst
                            | dateConst

(* Paths *)
<Paths>                     = <AnyPathConst> <PathsB>

<PathsB>                    = comma <AnyPathConst> <PathsB>
                            | EPSILON

<AnyPathConst> = pathConst

(* Runrules *)
<RunRules>                  = runRules <RulesList> semiColon

<RulesList>                 = ID <RulesListB>

<RulesListB>                = comma ID <RulesListB>
                            | EPSILON

(* Rules *)
<RuleDcls>                  = <RuleDcl> <RuleDclsB>

<RuleDclsB>                 = <RuleDcls>
                            | EPSILON

<RuleDcl>                   = rule ID lbrace <SelectProduction>
    <DoProduction> rbrace

(* Select *)
<SelectProduction>          = select <Selects> semiColon

<Selects>                   = <SelectExpression>

<SelectExpression>          = <SelectLogicalOr>

<SelectLogicalOr>           = <SelectLogicalAnd> <SelectLogicalOrB>

<SelectLogicalOrB>          = logicalOrOperator <SelectLogicalOr>
                            | EPSILON

<SelectLogicalAnd>          = <UnarySelectExpression> <SelectLogicalAndB>

<SelectLogicalAndB>         = logicalAndOperator <SelectLogicalAnd>
```

```
                              | EPSILON

<UnarySelectExpression>       = <Select>
                              | <UnarySelectOperator> <Select>

<UnarySelectOperator>         = boolNegation

<Select>                      = ID lparen <Values> rparen
                              | lparen <SelectLogicalOr> rparen

(* Do *)
<DoProduction>                = doSymbol <Dos> semiColon

<Dos>                         = <Do> <DosB>

<DosB>                        = comma <Do> <DosB>
                              | EPSILON

<Do>                          = move lparen <DoArgument> rparen
                              | rename lparen <DoArgument> rparen
                              | delete lparen <DoArgument> rparen
                              | copy lparen <DoArgument> rparen

<DoArgument>                  = <Expression>
                              | EPSILON

(* LIBRARY *)
<LibraryPart>                 = libraryColon <LibraryDcls>

<LibraryDcls>                 = <LibraryDcl> <LibraryDclsB>

<LibraryDclsB>                = <LibraryDcls>
                              | EPSILON

<LibraryDcl>                  = <SelectorDcl>
                              | <CalculatorDcl>

(* Selector *)
<SelectorDcl>                 = selector ID lparen <FormalParameterList> rparen
    lbrace <Stmts> rbrace

(* Calculator *)
<CalculatorDcl>               = calculator langlebracket <TypeName>
    ranglebracket ID lparen <FormalParameterList> rparen lbrace <Stmts> rbrace

(* Parameter list *)
<ActualParameterList>         = lparen <ActualParameterListB>

<ActualParameterListB>        = <Expression> <ActualParameterListC>
                              | rparen

<ActualParameterListC>        = comma <Expression> <ActualParameterListC>
                              | rparen

<FormalParameterList>         = <TypeName> ID <FormalParameterListB>
                              | EPSILON

<FormalParameterListB>        = comma <TypeName> ID <FormalParameterListB>
                              | EPSILON

(* Statements *)
<Stmts>                       = <Stmt> <StmtsB>

<StmtsB>                      = <Stmts>
                              | EPSILON

<Stmt>                        = <VariableDcl> semiColon
                              | <SelectedStmt> semiColon
                              | <ForEachFileStmt>
                              | <IfStmt>
```

```
                          | <AssignmentStmt> semiColon
                          | <BlockStmt>

<BlockStmt>               = lbrace <Stmts> rbrace

<VariableDcl>             = persistent <VariableDclB>
                          | <VariableDclB>

<VariableDclB>            = <TypeName> ID <VariableDclC>

<VariableDclC>            = assignmentOperator <Expression>
                          | EPSILON

<SelectedStmt>            = selected
                          | deSelected

<ForEachFileStmt>         = forEachFile <Stmt>

<IfStmt>                  = ifSymbol lparen <Expression> rparen <Stmt>
    <IfStmtB>

<IfStmtB>                 = elseSymbol <Stmt>
                          | EPSILON

<AssignmentStmt>          = ID assignmentOperator <Expression>

(* Expression *)
<Expression>              = <LogicalOrExpression>

<LogicalOrExpression>     = <LogicalAndExpression> <LogicalOrExpressionB>

<LogicalOrExpressionB>    = logicalOrOperator <LogicalOrExpression>
                          | EPSILON

<LogicalAndExpression>    = <EqualityExpression> <LogicalAndExpressionB>

<LogicalAndExpressionB>   = logicalAndOperator <LogicalAndExpression>
                          | EPSILON

<EqualityExpression>      = <RelationalExpression> <EqualityExpressionB>

<EqualityExpressionB>     = equalityOperator  <EqualityExpression>
                          | notEqualOperator <EqualityExpression>
                          | EPSILON

<RelationalExpression>    = <AdditiveExpression> <RelationalExpressionB>

<RelationalExpressionB>   = langlebracket  <RelationalExpression>
                          | ranglebracket <RelationalExpression>
                          | lesserEqualThanOperator <RelationalExpression>
                          | biggerEqualThanOperator <RelationalExpression>
                          | EPSILON


<AdditiveExpression>      = <MultiplicativeExpression> <AdditiveExpressionB>

<AdditiveExpressionB>     = minusOperator  <AdditiveExpression>
|plusOperator <AdditiveExpression>
                          | EPSILON

<MultiplicativeExpression>  = <UnaryExpression> <MultiplicativeExpressionB>

<MultiplicativeExpressionB> = multiplicationOperator
    <MultiplicativeExpression>
                          | divisionOperator <MultiplicativeExpression>
                          | moduloOperator <MultiplicativeExpression>
                          | EPSILON

<UnaryExpression>         = boolNegation <PrimaryExpression>
                          | minusOperator <PrimaryExpression>
```

```
                            |<PrimaryExpression>

<PrimaryExpression>         = <ExpressionValue>
                            | <IdentifierValue>
                            | lparen <Expression> rparen
                            | <ThisFile>
                            | undefined

<ExpressionValue>           = stringConst
                            | <BooleanConst>
                            | intConst
                            | <Time>
                            | pathConst
                            | regexConst

<IdentifierValue>           = ID <CalculatorCallRest>

<CalculatorCallRest>        = <ActualParameterList> | EPSILON

<ThisFile>                  = thisFile <ThisFileB>

<ThisFileB>                 = lparen metaDataSpecifierConst rparen
                            | dot <ThisFileC>

<ThisFileC>                 = isSelected
                            |  hashValue
```

Lexical elements D

This appendix shows the entire list of lexical elements of the PIMPL programming language. Each terminal is presented by the name according to the CFG, the associated regular expression and the terminals presentation in PIMPL, or examples of some. For the PIMPL presentation of the types, examples are shown, and these are separated by a comma.

Terminal	Regular expression	PIMPL presentation
lparen	/(/	(
rparen	/)/	)
lbrace	/{/	{
rbrace	/}/	}
langlebracket	/ < /	<
ranglebracket	/ > /	>
logicalOrOperator	/*OR*/	OR
logicalAndOperator	/*AND*/	AND
comma	/, /	,
semiColon	/; /	;
assignmentOperator	/ = /	=
equalityOperator	/ == /	==
dot	/ ./	.
lesserEqualThanOperator	/ <= /	<=
biggerEqualThanOperator	/ >= /	>=
minusOperator	/ − /	-
plusOperator	/ + /	+
multiplicationOperator	/ ∗ /	*
divisionOperator	///	/
moduloOperator	/%/	%
boolNegation	/*NOT*/	NOT

Table D.1. The entire list of the terminals, the associated regular expression and the presentation in PIMPL.

Terminal	Regular expression	PIMPL presentation
use	/*Use*/	Use
setWorkFolder	/*WorkFolder*/	WorkFolder
setRepeatOption	/*RunOption*/	RunOption
once	/*Once*/	Once
recurring	/*Recurring*/	Recurring
workInSubFolders	/*WorkInSubFolders*/	WorkInSubFolders
finallyOnChange	/*FinallyOnChange*/	FinallyOnChange
message	/*Message*/	Message
beep	/*Beep*/	Beep
exclude	/*Exclude*/	Exclude
persistent	/*Persistent*/	Persistent
forEachFile	/*ForeachFile*/	ForeachFile
selected	/*Selected*/	Selected
deSelected	/*Deselected*/	Deselected
ifSymbol	/*If*/	If
elseSymbol	/*Else*/	Else
runRules	/*RunRules*/	RunRules
doDcl	/*Do*/	Do
move	/*Move*/	Move
rename	/*Rename*/	Rename
delete	/*Delete*/	Delete
copy	/*Copy*/	Copy
includeLib	/*IncludeLib*/	IncludeLib
excludeLib	/*ExcludeLib*/	ExcludeLib
select	/*Select*/	Select
libraryColon	/*Library* : /	Library:
selector	/*Selector*/	Selector
calculator	/*Calculator*/	Calculator
thisFile	/*ThisFile*/	ThisFile
rule	/*Rule*/	Rule
isSelected	/*IsSelected*/	IsSelected
hashValue	/*HashValue*/	HashValue

Table D.2. The entire list of the terminals, the associated regular expression and the presentation in PIMPL(Continued).

Terminal	Regular expression	PIMPL presentation
set	/*Set*/	Set
intType	/*Int*/	Int
timeType	/*Time*/	Time
dateType	/*Date*/	Date
boolType	/*Bool*/	Bool
absolutePathType	/*AbsPath*/	AbsPath
relativePathType	/*RelPath*/	RelPath
absoluteFilePathType	/*AbsFilePath*/	AbsFilePath
relativeFilePathType	/*RelFilePath*/	RelFilePath
regexType	/*Regex*/	Regex
stringType	/*String*/	String
intConst	/[−]([0 − 9]) + /	42, 2014
stringConst	/"[^\/*\?\"|\:\<\>]*"/	"Inigo Montoya", "Hi!"
timeConst	/([01]?[0-9]|2[0-3]):[0-5][0-9]/	13:37, 16:15
dateConst	/[0-9]2(/|.|-)[0-9]2(/|.|-)[0-9]4|/	24/12/2013, 28/5/2014
absolutePathConst	/"[a-zA-Z]:\\([^\/*\?\"|\:\<\>]+\\)+"/	"C:\Users\InigoMontoya\Pictures\"
relativePathConst	/"*\\([^\/*\?\"|\:\<\>]+\\)+"/	"*\Pictures\Other\"
absoluteFilePathConst	/"[a-zA-Z]:\\([^\/*\?\"\:\:\<\>]+\\)+\.[^\/*\?\"\:\:\<\>]*"/	"C:\Users\InigoMontoya\ToDo.txt"
relativeFilePathConst	/"*\\([^\/*\?\"\:\:\<\>]+\\)+\.[^\/*\?\"\:\:\<\>]*"/	"*\InigoMontoya\ToDo.txt"
boolConst	/(*True*|*Yes*|*False*|*No*)/	True, False
ID	/[*a* − *zA* − *Z*]([*a* − *zA* − *Z*0 − 9]) ∗ /	Counter, x2
metaDataSpecifierConst	/'(~ ['']) +' /	'FileName'
undefined	/*Undefined*/	Undefined (See table 4.2)

Table D.3. The entire list of the terminals, the associated regular expression and the presentation in PIMPL(Continued).

Design and Syntax for elementary types

E

This appendix lists the types and operators for elementary types, which is not described in the other sections.

Elementary Types

This section contains the elementary types String, Int and Bool.

String

A String is a collection of characters, similar to other languages (like C# and Java), in terms of which characters are allowed. It is a collection of characters in a static order. A PIMPL string can contain all the letters allowed in files in Windows, this means all the unicode characters, except ", <, >, /, \, ?, |, :, # and *.

```
String ComputerName = VPCEB4X1E;
String DubName = ThisFile('FileName');
```

Listing E.1. The declaration of a String.

Listing E.1 shows an example of two different variables of a String type being declared and assigned values. The first is a String called ComputerName and the second is called DubName. DubName looks different because of the assignment to ThisFile('Name'). This is an indicator for the current files metadata for Name. ThisFile is described in section 4.4. A string can also be declared without being assigned to any value, then the value is undefined.

Int

An Int is an Integer and can only contain integers. The range of an Int is from -2^{31} until $2^{31} - 1$, as this is the range of the integer (Int32) for the targeted language C# [Microsoft MSDN, 2014a,b]. An Int can e.g. be used as parameters for Selectors and Calculators, as counting variables or as conditional variables in control statements. A decimal number would not be needed in PIMPL, as an Int is used for counting and conditions, which often is related to an exact number of files.

```
Int NumberOfPng = 0;
Int NumberOfJpeg = 0;

If (NumberOfPng < NumberOfJpeg) ...
```

Listing E.2. The declaration of an Int.

Listing E.2 shows the initialisation of an Int, and its use in a conditional statement, see section 4.7.1, for information about If-else statements. The listing shows the use of an Int in the library.

Boolean

Boolean values are allowed to be "True" or "False", but these can also be stated respectively as "Yes" and "No" in PIMPL. This is to familiarize the meaning for novice users. Boolean values are used when stating if the user wants to work in subfolders and can be used in e.g. conditional statements in the Library-part. An example of the use of Boolean values in a program is shown in listing E.3.

```
WorkInSubFolders Yes;   # Files in Subfolders are included #
```

Listing E.3. Example of use of a Boolean value.

Elementary Operators

This section contains the operators for the elementary types.

Boolean operators

There are five operators which can be used with the Boolean type. These can for example be used in an if-statement. The operators are shown below, and an example of use is shown in listing E.4.

1. a AND b
2. a OR b
3. NOT a
4. a == b
5. a != b

```
If (ThisFile('FileExtension') == ".jpg" OR ThisFile('FileExtension') !=
    ".png")
```

Listing E.4. The use of Boolean operators in a If-statement.

Three logical operators are included in PIMPL. These are AND, OR and NOT like in the C language. In PIMPL these operators are written in all uppercase letters, in order to make the readability clearer, by making the logical operators stand out from the rest of the code.

When operators are written with words instead of the three logic operators; "&", "|" and "!", the user can more easily read what it means, and while the operators stand out more.

The last two operators; equal(==) and not equal(!=), are used as comparison operators. With boolean types it is not possible to say if one expression is greater or smaller than another, therefore these are not included as boolean operators. Both the equal and not equal operator can be used in the Library-part, see section 4.7, and will therefore only be used by experienced users. The syntax in the Library-part is alike the C language, where '==' means equal(for certain types), because the single '=' is used as an assignment. '!=' is also taken from the C syntax and means not equal. In PIMPL the same applies for both operators.

Integers operators

There are six comparison operators and six arithmetic operators on integers, all 12 are listed below.

1. a < b
2. a > b
3. a <= b
4. a >= b
5. a == b
6. a != b
7. a + b
8. a - b
9. a * b
10. a / b
11. a % b
12. -a

The operators have the syntax as in the C language. The comparison operators are shown as the first six operators in the above list. These six operators can be used in the conditional part of an if-statement, see section 4.7.1 for an example, as the result of the expressions is a boolean type. The arithmetic operators are shown as the last six operators in the above list. The division operator(/) is rounding the value, depending on the value. The rounding rules is the same as the C# rules, and here values with values .5 will be rounded down, and values above is rounded up.

In the C language, more operators exists, like increment and decrement, but these have been chosen not to be included, as their task can be fulfilled differently. Increment and decrement in expressions gives side effects, that may decrease reliability of a PIMPL program.

String operators

There are three operators on strings in PIMPL, as seen in the following list.

1. a == b
2. a != b
3. a + b

The two first operators are comparison operators and can therefore be used in conditional statements, like an if-statement, as the comparison operators on integers. On Strings it is not possible to use the smaller- or greater-than operator and the equal operator(==) only results in true, if an only if the Strings are exactly equal. This also means that the comparison is case sensitive. RegEx can be used if the user wants to compare strings without taking case sensitivity into consideration. The last operator, plus(+), is used to concatenate two strings, this makes it possible to build new strings from existing

strings. As an example, a file name that gets its creation date added to the name. See the example in listing E.5, and note that ThisFile('..') gives a string based on "Name" and "CreationDate" metadata.

```
String newName = ThisFile('DateCreated') + ThisFile('FileName');
```

Listing E.5. The concatenation of two Strings.

Example of a small PIMPL program

F

In this chapter, it is shown how a small example PIMPL program is running, step-by-step. This is to get an idea and overview of how PIMPL works. The small example program contains two separate files, a Rule-part shown on listing F.1 and a Library-part shown on listing F.2.

```
Use "StandardLib.lib";
WorkFolder "C:\Users\username\desktop\test\";
RunOption Once;
WorkInSubFolders No;

Rule ToOrganizeImages
{
    Select Type(".jpg") AND TakenBetweenTwoDates(01-01-2013, 01-01-2014);
    Do Move("C:\Users\username\desktop\2013\" + ThisFile('Photo+DateTaken') +
        "\\");
}
```

Listing F.1. The Rule-part of the example.

```
Library:

Selector Type(String s)
{
    If(ThisFile('FileExtension') == s)
        Selected;
}

Selector TakenBetweenTwoDates(Date d1, Date d2)
{
    Date actualDate = ThisFile('Photo+DateTaken');

    If(actualDate >= d1 AND actualDate <= d2)
        Selected;
}
```

Listing F.2. The Library-part of the example.

Folder containing these files

- A: 30. Maj 2008
- B: 23. Februar 2013
- C: 8. April 2013
- D: 27. April 2013

- E: 2. Juni 2013
- F: 12. Juni 2013
- G: 17. Juli 2013

Instructions
All pictures from 2013, should be moved to a folder named "2013". In the 2013 folder, subfolders are specified with the date the pictures was captured.

Rule-part review

```
1 Use "StandardLib.lib";
2 WorkFolder "C:\Users\username\desktop\test\";
3 RunOption Once;
4 WorkInSubFolders No;
```

Listing F.3. An example of some options.

Listing F.3 show four lines of code containing the specified options. First line define which library to use. This file have to be located in the exact folder as the program when the program is compiled. Second line is where the WorkFolder is defined. In this example, it is the folder where the seven pictures are located. Third line define if the program should run in the background and keep the folder updated or if the program only runs once. In this case, it is defined to run once. Fourth line define if the program should work in subfolders or ignore these. In this example, subfolders are ignored.

```
1 Rule ToOrganizeImages
2 {
3     Select Type(".jpg") AND TakenBetweenTwoDates(01-01-2013, 01-01-2014);
4     Do Move("C:\Users\username\desktop\2013\" + ThisFile('Photo+DateTaken') +
          "\\");
5 }
```

Listing F.4. An example of a Rule.

Listing F.4 shows the declaration of a rule called "ToOrganizeImages". The rule specify that files of the type .jpg should be selected. Type() is in this example a Selector from the Library. The rule also specify that the images wanted, os taken between 01-01-2013 and 31-12-2013. TakenBetweenTwoDates() is a Selector from the Library.

The selected files have to be moved to a new folder named "2013". This new folder have to contain subfolders named after the date taken from the pictures. If the folders does not exist, they are created.

Library-part review
At the start of a Library-file, "Library:" is written, to define that it is a Library. Afterwards a Selector named "Type" is written. The "Type" takes a String as input parameter. The Selector then choses files based on the file's 'FileExtension' metadata. This part of the Library is shown in listing F.5.

```
Library:

Selector Type(String s)
{
    If(ThisFile('FileExtension') == s)
        Selected;
}
```

Listing F.5. The Library-part of the example.

In Type, an If-statement is used to compare the file's metadata named type with the string s. If these match, then the file is marked as selected. If they do not match, they are deselected. By default, all files are marked as deselected.

```
Selector TakenBetweenTwoDates(Date d1, Date d2)
{
    Date actualDate = ThisFile('Photo+DateTaken');

    If(actualDate >= d1 AND actualDate <= d2)
        Selected;
}
```

Listing F.6. The TakenBetweenTwoDates Selector used in the example.

The Selector named "TakenBetweenTwoDates" take 2 parameters of the type Date, named d1 and d2. In "TakenBetweenTwoDates", it starts by making a temp variable named actualdate of the type Date. The actual file's metadata for 'Photo+DateTaken' is assigned to actualdate. Actualdate is then used in an if-statement to check if actualdate is between d1 and d2. If actualdate is between d1 and d2, then the actual file is marked as selected.

Step by step walk-through of execution

Overall process

When executing the program, it start by executing the first rule onto the files located in the folder C:\users\username\desktop\test\. This means that the folder will be run through, one file at a time, and mark which files the file operations should be performed. When all the files has been checked, operations on the selected files are executed. After the first rule is executed, the next rule executes, which executes with the same process as the first. When all rules have finished executing, the program terminates, as the RunOption is defined to "once".

How files are chosen

To choose a file, the conditions in the Select has to be fulfilled. The conditions are read from left to right and if the first condition is true, it checks the next condition if the AND keyword is used. The AND keyword defines that all conditions connected with AND have to be fulfilled for at file to be chosen.

Concrete example of execution

File A is being checked if the type is .jpg. The first condition is true and file A will be checked on the second condition. The picture is taken in 2008 and not in 2013, therefore

the second condition is not fulfilled and the file is still marked as deselected. The program then checks on file B. B is of type .jpg and first condition is fulfilled. B is taken in 2013 and therefore fulfill second condition and is marked as selected. This process is repeated for the last five files in the folder.

When all the files have been run through, the operations have to be executed. The first selected file, which is B, is moved to the folder C:\Users\username\desktop\2013\23-02-2013\. This folder do not exist and is therefore created before the file is moved. This process is then repeated for the remaining chosen files, although this time the same folder called "2013" is used, as it now exist.

Structural Operational Semantics

A more precise informal description of *RunOption* and *FinallyOnChange* is presented here.

RunOption
A PIMPL program must include a *RunOption* option statement which is used to define which of two modes a PIMPL program can run in. *RunOptionOnce*; means that the PIMPL program is executed once i.e. that all the rules are executed once if no RunRules option declaration has been made. Similarly all the rules referenced in a RunRules option declaration are executed once if such a declaration is made.

RunOption Recurring; means the that the rules of the program are first executed once in the same way as if *RunOption* was defined as *RunOptionOnce*;. But the program does not terminate after the first execution of the given rules. The program places a hook on its workfolder in the file system and waits for changes in the workfolder. The rules are then executed again every time the program is notified, by the filesystem, of changes in the workfolder.

FinallyOnChange
It is possible to include a *FinallyOnChange* option statement in the start of a PIMPL program rule file. This option specifies that user must be notified if changes occurs in workfolder which the program is working in. *FinallyOnChange Message*(*s*); specifies that the user must be notified with a message *s* if changes have occurred, i.e. the rules of the PIMPL program has invoked some do operation on a file in the workfolder. *FinallyOnChange Beep*(); specifies that the user must be notified with a sound if changes have occurred.

The rest of the appendix states the Structural Operational Semantics (SOS) transition rules and the Type rules for PIMPL. Note that it does not cover all aspects of the PIMPL language.

SOS Transition rules

Expression

[*INT*]	$n \rightarrow_e nv$	$if\ \mathcal{N}[\![n]\!] = nv$
[*STRING*]	$s \rightarrow_e sv$	$if\ \mathcal{S}[\![s]\!] = sv$
[*BOOL*]	$b \rightarrow_e bv$	$if\ \mathcal{B}[\![b]\!] = bv$
[*PATH*]	$p \rightarrow_e pv$	$if\ \mathcal{P}[\![p]\!] = pv$
[*TIME*]	$t \rightarrow_e tv$	$if\ \mathcal{T}[\![t]\!] = tv$
[*DATE*]	$d \rightarrow_e dv$	$if\ \mathcal{D}[\![d]\!] = dv$
[*REGEX*]	$re \rightarrow_e rev$	$if\ \mathcal{RE}[\![re]\!] = rev$
[*MSSPEC*]	$ms \rightarrow_e msv$	$if\ \mathcal{MS}[\![ms]\!] = msv$
$[DIT - TO - S]$	$ditv \rightarrow_e sv$	$if\ \mathcal{DITTOS}[\![ditv]\!] = sv$
$[S - TO - RE]$	$sv \rightarrow_e rev$	$if\ \mathcal{STOREG}[\![sv]\!] = rev$
[*ALL*]	$all \rightarrow_e allv$	$if\ \mathcal{ALL}[\![all]\!] = allv$

Table G.1. Big step operational semantics transition rules for primary expressions.

[*PLUS*]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e nv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e nv_2}{env_{ID},\ sto,\ env_F \vdash e_1 + e_2 \rightarrow_e nv}$ *where* $nv = nv_1 + nv_2$
[*MINUS*]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e nv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e nv_2}{env_{ID},\ sto,\ env_F \vdash e_1 - e_2 \rightarrow_e nv}$ *where* $nv = nv_1 - nv_2$
[*MULT*]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e nv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e nv_2}{env_{ID},\ sto,\ env_F \vdash e_1 * e_2 \rightarrow_e nv}$ *where* $nv = nv_1 \cdot nv_2$
[*DIV*]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e nv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e nv_2}{env_{ID},\ sto,\ env_F \vdash e_1/e_2 \rightarrow_e nv}$ *where* $nv = \dfrac{nv_1}{nv_2}$
[*MOD*]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e nv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e nv_2}{env_{ID},\ sto,\ env_F \vdash e_1 \% e_2 \rightarrow_e nv}$ *where* $nv = nv_1$ **mod** nv_2
[*UMINUS*]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e \rightarrow_e nv_1}{env_{ID},\ sto,\ env_F \vdash -e \rightarrow_e nv}$ *where* $nv = -nv_1$
[*PARENT*]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e \rightarrow_e allv}{env_{ID},\ sto,\ env_F \vdash (e) \rightarrow_e allv}$

Table G.2. Big-step SOS transition rules for arithmetic expressions.

$[CON - S]$	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e sv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e sv_2}{env_{ID},\ sto,\ env_F \vdash e_1 + e_2 \rightarrow_e sv}$ $where\ sv = sv_1 \circ sv_2$
$[CON - P - S]$	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e pv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e sv}{env_{ID},\ sto,\ env_F \vdash e_1 + e_2 \rightarrow_e pv}$ $where\ pv = SToPath[PathToS[pv_1] \circ\ sv]$ $and\ SToPath:\ sv \longrightarrow\ pv$ $and\ PathToS:\ pv \longrightarrow\ sv$
$[CON - RE]$	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e rev_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e rev_2}{env_{ID},\ sto,\ env_F \vdash e_1 + e_2 \rightarrow_e rev}$ $where\ rev = rev_1 \circ rev_2$

Table G.3. Big-step SOS transition rules for concatenation expression.

[$GTtrue$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e ditv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e ditv_2}{env_{ID},\ sto,\ env_F \vdash e_1 > e_2 \rightarrow_e \top}$ $if\ ditv_1 > ditv_2$
[$GTfalse$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e ditv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e ditv_2}{env_{ID},\ sto,\ env_F \vdash e_1 > e_2 \rightarrow_e \bot}$ $if\ ditv_1 \leq ditv_2$
[$GTEQtrue$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e ditv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e ditv_2}{env_{ID},\ sto,\ env_F \vdash e_1 >= e_2 \rightarrow_e \top}$ $if\ ditv_1 \geq ditv_2$
[$GTEQfalse$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e ditv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e ditv_2}{env_{ID},\ sto,\ env_F \vdash e_1 >= e_2 \rightarrow_e \bot}$ $if\ ditv_1 < ditv_2$
[$LTtrue$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e ditv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e ditv_2}{env_{ID},\ sto,\ env_F \vdash e_1 < e_2 \rightarrow_e \top}$ $if\ ditv_1 < ditv_2$
[$LTfalse$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e ditv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e ditv_2}{env_{ID},\ sto,\ env_F \vdash e_1 < e_2 \rightarrow_e \bot}$ $if\ ditv_1 \geq ditv_2$
[$LTEQtrue$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e ditv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e ditv_2}{env_{ID},\ sto,\ env_F \vdash e_1 <= e_2 \rightarrow_e \top}$ $if\ ditv_1 \leq ditv_2$
[$LTEQfalse$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e ditv_1 \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e ditv_2}{env_{ID},\ sto,\ env_F \vdash e_1 <= e_2 \rightarrow_e \bot}$ $if\ ditv_1 > ditv_2$

Table G.4. Big-step SOS transition rules for comparison operators.

[*EQUALtrue*]	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e allv_1 \quad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e allv_2}{env_{ID},\ sto,\ env_F \vdash e_1 == e_2 \rightarrow_e \top}$ $if\ allv_1 = allv_2$ $and\ allv_1 \notin \mathbb{RE} \qquad and\ allv_2 \notin \mathbb{RE}$
[*NOTEQUALtrue*]	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e allv_1 \quad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e allv_2}{env_{ID},\ sto,\ env_F \vdash e_1 != e_2 \rightarrow_e \top}$ $if\ allv_1 \neq allv_2$ $and\ allv_1 \notin \mathbb{RE} \qquad and\ allv_2 \notin \mathbb{RE}$
[*MATCHEDLefttrue*]	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e rev \quad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e allv}{env_{ID},\ sto,\ env_F \vdash e_1 == e_2 \rightarrow_e \top}$ $if\ \mathcal{TOS}(allv) \in L(rev)$ $and\ allv \notin \mathbb{RE}$
[*MATCHEDRighttrue*]	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e allv \quad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e rev}{env_{ID},\ sto,\ env_F \vdash e_1 == e_2 \rightarrow_e \top}$ $if\ \mathcal{TOS}(allv) \in L(rev)$ $and\ allv \notin \mathbb{RE}$
[*NO − MATCHLefttrue*]	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e rev \quad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e allv}{env_{ID},\ sto,\ env_F \vdash e_1 != e_2 \rightarrow_e \top}$ $if\ \mathcal{TOS}(allv) \notin L(rev)$ $and\ allv \notin \mathbb{RE}$
[*NO − MATCHRighttrue*]	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e allv \quad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e rev}{env_{ID},\ sto,\ env_F \vdash e_1 != e_2 \rightarrow_e \top}$ $if\ \mathcal{TOS}(allv) \notin L(rev)$ $and\ allv \notin \mathbb{RE}$

Table G.5. Big-step SOS transition rules for equality operators.

[$EQUALfalse$]	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e allv_1 \quad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e allv_2}{env_{ID},\ sto,\ env_F \vdash e_1 == e_2 \rightarrow_e \bot}$ $if\ allv_1 \neq allv_2$ $and\ allv_1 \notin \mathbb{RE} \qquad and\ allv_2 \notin \mathbb{RE}$
[$NOTEQUALfalse$]	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e allv_1 \quad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e allv_2}{env_{ID},\ sto,\ env_F \vdash e_1 != e_2 \rightarrow_e \bot}$ $if\ allv_1 = allv_2$ $and\ allv_1 \notin \mathbb{RE} \qquad and\ allv_2 \notin \mathbb{RE}$
[$MATCHEDLeftfalse$]	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e allv \quad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e rev}{env_{ID},\ sto,\ env_F \vdash e_1 == e_2 \rightarrow_e \bot}$ $if\ \mathcal{TOS}(allv) \notin L(rev)$ $and\ allv \notin \mathbb{RE}$
[$MATCHEDRightfalse$]	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e rev \quad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e allv}{env_{ID},\ sto,\ env_F \vdash e_1 == e_2 \rightarrow_e \bot}$ $if\ \mathcal{TOS}(allv) \notin L(rev)$ $and\ allv \notin \mathbb{RE}$
[$NO-MATCHLeftfalse$]	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e allv \quad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e rev}{env_{ID},\ sto,\ env_F \vdash e_1 != e_2 \rightarrow_e \bot}$ $if\ \mathcal{TOS}(allv) \in L(rev)$ $and\ allv \notin \mathbb{RE}$
[$NO-MATCHRightfalse$]	$\frac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e rev \quad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e allv}{env_{ID},\ sto,\ env_F \vdash e_1 != e_2 \rightarrow_e \bot}$ $if\ \mathcal{TOS}(allv) \in L(rev)$ $and\ allv \notin \mathbb{RE}$

Table G.6. Big-step SOS transition rules for equality operators.

[$ORtrue$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_i \rightarrow_e \top}{env_{ID},\ sto,\ env_F \vdash e_1\ OR\ e_2 \rightarrow_e \top}$ $i \in \{1, 2\}$
[$ORfalse$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e \bot \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e \bot}{env_{ID},\ sto,\ env_F \vdash e_1\ OR\ e_2 \rightarrow_e \bot}$
[$ANDtrue$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e \top \qquad env_{ID},\ sto,\ env_F \vdash e_2 \rightarrow_e \top}{env_{ID},\ sto,\ env_F \vdash e_1\ AND\ e_2 \rightarrow_e \top}$
[$ANDfalse$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e_i \rightarrow_e \bot}{env_{ID},\ sto,\ env_F \vdash e_1\ AND\ e_2 \rightarrow_e \bot}$ $i \in \{1, 2\}$
[$NOTtrue$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e \rightarrow_e \bot}{env_{ID},\ sto,\ env_F \vdash NOTe \rightarrow_e \top}$
[$NOTfalse$]	$\dfrac{env_{ID},\ sto,\ env_F \vdash e \rightarrow_e \top}{env_{ID},\ sto,\ env_F \vdash NOTe \rightarrow_e \bot}$

Table G.7. Big-step SOS transition rules for logical operators.

$[ID - VAR]$	$env_{ID},\ sto,\ env_F \vdash id \rightarrow_e allv$ $if\ env_{ID}\ id = l$ $and\ sto\ l = allv$
$[ID - CALC]$	$env_{ID},\ sto,\ env_F \vdash e_1 \rightarrow_e allv_1,\ \ldots, e_n \rightarrow_e allv_n$ $\langle S,\ env'_{ID}[Result \mapsto l_0,\ id_1 \mapsto l_1,\ \ldots,\ id_n \mapsto l_n],\ sto[l_0 \mapsto Default(T),\ l_1 \mapsto allv_1,\ \ldots,\ l_n \mapsto allv_n],\ env_F\rangle \rightarrow_s$ $\langle\ env''_{ID},\ sto',\ env_F\rangle$ $env''_{ID},\ sto' \vdash Result \rightarrow_e allv_R$ $\overline{env_{ID}, sto,\ env_F \vdash id(e_1,\ \ldots, e_n) \rightarrow_e allv_R}$ $where\ env_{ID}\ id = l$ $and\ sto\ l = (S, T, (id_1,\ \ldots, id_n), env'_{ID})$
$[CALL - S]$	$all_1 \rightarrow_e allv_1,\ \ldots, all_n \rightarrow_e allv_n$ $\langle S, env'_{ID}[\_IsSelected \mapsto l',\ \_filepath \mapsto l'',\ id_1 \mapsto l_1,\ \ldots,\ id_n \mapsto l_n],\ sto[l' \mapsto \bot,\ l'' \mapsto currentFilePath,\ l_1 \mapsto allv_1,\ \ldots,\ l_n \mapsto allv_n],\ env_F\rangle \rightarrow_s \langle env''_{ID}, sto',\ env_F\rangle$ $env''_{ID},\ sto' \vdash \_IsSelected \rightarrow_e bv$ $\overline{env_{ID},\ sto,\ env_F \vdash id(all_1,\ \ldots, all_n) \rightarrow_e bv}$ $where\ env_{ID}\ id = l$ $and\ sto\ l = (S, Bool,\ (id_1,\ \ldots, id_n),\ env'_{ID})$ $and\ sto(\ env_{ID}(\_filepath)) = (currentFilePath,\ filedata,\ mstov,\ bv_{old})$

Table G.8. Big-step SOS transition rules for ID expressions.

	$$\frac{ms \rightarrow_e msv}{env_{ID},\ sto,\ env_F \vdash ThisFile(ms) \rightarrow_e allv}$$
$[TF - MS]$	$where\ sto(env_{ID}\ \_filepath) = pv$ $and\ env_F\ pv = (pv,\ filedata,\ mstov,\ bv)$ $and\ allv = mstov(msv)$
$[TF - ISSELECTED]$	$env_{ID},\ sto,\ env_F \vdash ThisFile.IsSelected \rightarrow_e bv$ $where\ sto(env_{ID}\ \_filepath) = pv$ $and\ env_F\ pv = (pv,\ filedata,\ mstov,\ bv)$
$[TF - HASHVALUE]$	$env_{ID},\ sto,\ env_F \vdash ThisFile.HashValue \rightarrow_e sv$ $where\ sto(env_{ID}\ \_filepath) = pv$ $andsv = GetHashValue(pv)$

Table G.9. Big-step operational semantics transition rules for ThisFile expressions.

Statement

$[SELECTED]$	$\langle Selected;,\ env_{ID},\ sto,\ env_F\rangle \longrightarrow_S$ $(env_{ID}[\_IsSelected \mapsto l],\ sto[l \mapsto \top],\ env_F)$
$[DESELECTED]$	$\langle Deselected;,\ env_{ID},\ sto,\ env_F\rangle \longrightarrow_S$ $(env_{ID}[\_IsSelected \mapsto l],\ sto[l \mapsto \bot],\ env_F)$

Table G.10. Operational semantics for selected statements.

$[DCL - VAR]$	$\langle T\ id;,\ env_{ID},\ sto,\ env_F\rangle \longrightarrow_S$ $(env_{ID}[id \mapsto l],\ sto[l \mapsto Default(T)],\ env_F)$
$[DCL - VAR - INIT]$	$\dfrac{env_{ID},\ sto,\ env_F \vdash e \rightarrow_e allv}{\langle T\ id\ =\ e;,\ env_{ID},\ sto,\ env_F\rangle \longrightarrow_S (env_{ID}[id \mapsto l],\ sto[l \mapsto allv],\ env_F)}$
$[DCL - CALC]$	$\langle Calculator < T > id(T_1\ id_1,\ ...\ T_n\ id_n)\{\ S\ \},\ env_{ID},\ sto,\ env_F\rangle \longrightarrow_S$ $(env_{ID}[id \mapsto l,\ sto[l \mapsto (S, T,\ (id_1,\ ...\ , id_n)],\ env_{ID})],\ env_F)$ $where\ \exists S^{'} \in StmtExpand(S)\ AND\ S^{'}\ =\ Result\ =\ e;$
$[DCL - S]$	$\langle Selector\ id(T_1\ id_1,\ ...\ T_n\ id_n)\{\ S\ \},\ env_{ID},\ sto,\ env_F\rangle \longrightarrow_S$ $(env_{ID}[\ id \mapsto l],\ sto[l \mapsto (S, Bool,\ (id_1,\ ...\ , id_n),\ env_{ID})]\ env_F)$
$[DCL - R]$	$\langle Rule\ id\{Select\ e; Do\ S_1, S_2,\ ...\ ,\ S_n;\},\ env_{ID},\ sto,\ env_F\rangle \longrightarrow_S$ $(env_{ID}[\ id \mapsto l],\ sto[l \mapsto (S,\ ok,\ (),\ env_{ID})],\ env_F)$ $where\ S = ForeachFile\ If(e)\{\ S_1\ S_2\ \ ...\ \ S_n\}$

Table G.11. Operational semantics for declarations.

$[RUNR]$	$\dfrac{\langle S_{Dcls},\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle env'_{ID},\ sto',\ env_F\rangle \quad \langle S_{Rules},\ env'_{ID},\ sto',\ env_F\rangle \rightarrow_s \langle env'_{ID},\ sto'',\ env'_F\rangle}{\langle RunRules\ \ id_1,\ ...\ , id_n;\ S_{Dcls},\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle env'_{ID},\ sto'',\ env'_F\rangle}$ $where\ (sto'(env'_{ID}\ id_1) = (S_1,\ ok,\ (),\ env^1_{ID}))\ ...\ (sto(env_{ID}\ id_n) =$ $(S_n,\ ok,\ (),\ env^n_{ID}))$ $and\ S_{Rules} =\ S_1\ S_2\ ...\ S_n$

Table G.12. Big-step SOS transition rules for Rules.

$$\frac{\langle S, env'_{ID},\ sto'',\ env_F\rangle \rightarrow_s \langle env'_{ID},\ sto^3,\ env''_F\rangle \quad \langle ForeachFileIt\ S,\ env''_{ID},\ sto^3,\ env''_F\rangle \rightarrow_s \langle env''_{ID},\ sto',\ env'_F\rangle}{\langle ForeachFile\ S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto',\ env'_F\rangle}$$

$[FFILE - F - S]$

$$\begin{aligned}
&where\ nextFile(firstFile) = (firstFilePath,\ filedata,\ mstov,\ bv)\\
&and\ env''_{ID} = env_{ID}[_implicitFileNestingLevel \mapsto l']\\
&and\ sto^4 = sto[l' \mapsto 1]\\
&and\ id = CurrentFileName(env''_{ID}, sto^4)\\
&and\ env'_{ID} = env''_{ID}[id \mapsto l]\\
&and\ sto'' =\ sto^4[l \mapsto firstFilePath]\\
&if(\ env_{ID}(_implicitFileNestingLevel) \uparrow {}^1)\\
&\ and((\ env_{ID}(id) \uparrow))
\end{aligned}$$

$$\frac{\langle S, env'_{ID},\ sto'',\ env_F\rangle \rightarrow_s \langle env'_{ID},\ sto^3,\ env''_F\rangle \quad \langle ForeachFileIt\ S,\ env''_{ID},\ sto^3,\ env''_F\rangle \rightarrow_s \langle env''_{ID},\ sto',\ env'_F\rangle}{\langle ForeachFile\ S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto',\ env'_F\rangle}$$

$[FFILE - S]$

$$\begin{aligned}
&where\ nextFile(firstFile) = (firstFilePath,\ filedata,\ mstov,\ bv)\\
&and\ env''_{ID} = env_{ID}[_implicitFileNestingLevel \mapsto l']\\
&and\ sto^4 = sto[l' \mapsto sto(env_{ID}(_implicitFileNestingLevel)) + 1]\\
&and\ id = CurrentFileName(env''_{ID}, sto^4) and id \neq CurrentFileName(env_{ID}, sto)\\
&and\ env'_{ID} = env''_{ID}[id \mapsto l]\\
&and\ sto'' =\ sto^4[l \mapsto firstFilePath]
\end{aligned}$$

Table G.13. Big-step SOS transition rules for ForeachFile starts.

[1] "func(b)↑" is understood as "f(b) is undefined"

$$\frac{\begin{array}{l}\langle S, env'_{ID},\ sto'',\ env_F\rangle \rightarrow_s \langle env'_{ID},\ sto^3,\ env''_F\rangle \\ \langle ForeachFileIt\ \ S,\ \ env'_{ID},\ \ sto^3,\ \ env''_F\rangle \ \ \rightarrow_s \\ \langle env'_{ID},\ sto',\ env'_F\rangle\end{array}}{\langle ForeachFileIt\ S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto',\ env'_F\rangle}$$

$[FFILE - I]$

$where\ id = CurrentFileName(env_{ID}, sto)$
$and\ nextFile(env_F(sto(env_{ID}(id)))) = (currentFilePath,\ filedata,\ mstov,\ bv)$
$and\ env'_{ID} = env_{ID}[id \mapsto l]$
$and\ sto'' =\ sto[l \mapsto currentFilePath]$

$\langle ForeachFile\ S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto,\ env_F\rangle$

$[FFILE - EMPTY]$

$if(nextFile(firstFile) = fileListEnd)$

$\langle ForeachFileIt\ S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto,\ env_F\rangle$

$[FFILE - END]$

$if(\ nextFile(env_F(sto(env_{ID}(CurrentFileName(env_{ID}, sto))))) = fileListEnd)$

Table G.14. Big-step SOS transition rules for ForeachFile iteration and end.

[*IFELSEtrue*]	$\dfrac{\langle S_1, env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle env'_{ID},\ sto'\ env'_F\rangle}{\langle If(e)\ S_1\ Else\ S_2,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ env'_F\rangle}$ $if\ env_{ID},\ sto,\ env_F \vdash e \rightarrow_e \top$
[*IFELSEfalse*]	$\dfrac{\langle S_2, env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle env'_{ID},\ sto'\ env'_F\rangle}{\langle If(e)\ S_1\ Else\ S_2,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle env_{ID},\ env'_F\rangle}$ $if\ env_{ID},\ sto,\ env_F \vdash e \rightarrow_e \bot$
[*IFtrue*]	$\dfrac{\langle S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env'_{ID},\ sto,\ env'_F\rangle}{\langle If(e)\ S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env'_{ID},\ sto,\ env'_F\rangle}$ $if\ env_{ID},\ sto,\ env_F \vdash e \rightarrow_e \top$
[*IFfalse*]	$\langle If(e)\ S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_e \langle\ env_{ID},\ sto,\ env_F\rangle$ $if\ env_{ID},\ sto,\ env_F \vdash e \rightarrow_e \bot$
[*Block*]	$\dfrac{\langle S,\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env'_{ID},\ sto',\ env'_F\rangle}{\langle\{S\},\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto'\ env'_F\rangle}$

Table G.15. Big-step SOS transition rules for control structures.

$$\frac{e \rightarrow_e pv_1}{\langle Move(e),\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto,\ env'_F\rangle}$$

$[DO - MOVE]$

$where\ env_F(sto(env_{ID}(\_filePath))) = (pv_2,\ filedata,\ mstov,\ bv)$
$and\ env''_F = env_F[sto(env_{ID}(\_filePath)) \mapsto \epsilon]$
$and\ env'_F = env''_F[pv_1 \mapsto (pv_1,\ filedata,\ mstov,\ bv)]$
$if\ sto(env_{ID}(\_filepath)) \neq fileListEnd$

$$\frac{e \rightarrow_e pv}{\langle Copy(e),\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto,\ env_F[pv \mapsto (pv,\ filedata,\ mstov, \bot)]\rangle}$$

$[Do - Copy]$

$where\ env_F(sto(env_{ID}(\_filePath))) = (pv',\ filedata,\ mstov,\ bv)$
$if\ sto(env_{ID}(\_filepath)) \neq fileListEnd$

$$\frac{e \rightarrow_e sv}{\langle Rename(\ e),\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto,\ env'_F[pv' \mapsto (pv',\ filedate,\ mstov, \bot)]\rangle}$$

$[Do - Rename]$

$where\ env_F(sto(env_{ID}(\_filePath))) = (pv,\ filedata,\ mstov,\ bv)$
$and\ env'_F = env_F[sto(env_{ID}(\_filePath)) \mapsto \epsilon]$
$pv' = ReplaceFileName(pv, sv)$
$if\ sto(env_{ID}(\_filepath)) \neq fileListEnd$

$\langle Delete(),\ env_{ID},\ sto,\ env_F\rangle \rightarrow_s \langle\ env_{ID},\ sto,\ env_F[pv \mapsto \epsilon]\rangle$

$[Do - Delete]$

$where\ env_F(sto(env_{ID}(\_filePath))) = (pv,\ filedata,\ mstov,\ bv)$
$if\ sto(env_{ID}(\_filepath)) \neq fileListEnd$

Table G.16. Big-step SOS transition rules for do statements.

Type rules

Expressions

$[INT_{EXP}]$	$E \vdash n \rightarrow_e Int$
$[STRING_{EXP}]$	$E \vdash s \rightarrow_e String$
$[BOOL_{EXP}]$	$E \vdash b \rightarrow_e Bool$
$[PATH_{EXP}]$	$E \vdash p \rightarrow_e Path$
$[TIME_{EXP}]$	$E \vdash t \rightarrow_e Time$
$[DATE_{EXP}]$	$E \vdash d \rightarrow_e Date$
$[REGEX_{EXP}]$	$E \vdash re \rightarrow_e RegEx$

Table G.17. Type rules for primary expressions.

$[PLUS_{EXP}]$	$\frac{E \vdash e_1 : Int \qquad E \vdash e_2 : Int}{E \vdash e_1 + e_2 : Int}$
$[MINUS_{EXP}]$	$\frac{E \vdash e_1 : Int \qquad E \vdash e_2 : Int}{E \vdash e_1 - e_2 : Int}$
$[MULT_{EXP}]$	$\frac{E \vdash e_1 : Int \qquad E \vdash e_2 : Int}{E \vdash e_1 * e_2 : Int}$
$[DIV_{EXP}]$	$\frac{E \vdash e_1 : Int \qquad E \vdash e_2 : Int}{E \vdash e_1 / e_2 : Int}$
$[MOD_{EXP}]$	$\frac{E \vdash e_1 : Int \qquad E \vdash e_2 : Int}{E \vdash e_1 \% e_2 : Int}$
$[PARENT_{EXP}]$	$\frac{E \vdash e : B}{E \vdash (e) : B}$
$[BOOL\_NEGATION_{EXP}]$	$\frac{E \vdash e : B}{E \vdash NOT\ e : B}$
$[INT\_NEGATION_{EXP}]$	$\frac{E \vdash e : Int}{E \vdash -e : Int}$

Table G.18. Type rules for arithmetic.

$[CON - S_{EXP}]$	$\frac{E \vdash e_1 : String \qquad E \vdash e_2 : String}{E \vdash e_1 + e_2 : String}$
$[CON - P - S_{EXP}]$	$\frac{E \vdash e_1 : Path \qquad E \vdash e_2 : String}{E \vdash e_1 + e_2 : Path}$
$[CON - REGEX_{EXP}]$	$\frac{E \vdash e_1 : RegEx \qquad E \vdash e_2 : RegEx}{E \vdash e_1 + e_2 : RegEx}$

Table G.19. Type rules for concatenation.

$[GREATERTHAN_{EXP}]$	$\frac{E \vdash e_1 : NT \qquad E \vdash e_2 : NT}{E \vdash e_1 > e_2 : Bool}$
$[GREATERTHANEQ_{EXP}]$	$\frac{E \vdash e_1 : NT \qquad E \vdash e_2 : NT}{E \vdash e_1 \geq e_2 : Bool}$
$[LESSERTHAN_{EXP}]$	$\frac{E \vdash e_1 : NT \qquad E \vdash e_2 : NT}{E \vdash e_1 < e_2 : Bool}$
$[LESSERTHANEQ_{EXP}]$	$\frac{E \vdash e_1 : NT \qquad E \vdash e_2 : NT}{E \vdash e_1 \leq e_2 : Bool}$
$[EQUAL_{EXP}]$	$\frac{E \vdash e_1 : T \qquad E \vdash e_2 : T}{E \vdash e_1 == e_2 : Bool}$
$[NOTEQUAL_{EXP}]$	$\frac{E \vdash e_1 : T \qquad E \vdash e_2 : T}{E \vdash e_1 ! = e_2 : Bool}$

Table G.20. Type rules for comparison operators.

$[OR_{EXP}]$	$\frac{E \vdash e_1 : Bool \qquad E \vdash e_2 : Bool}{E \vdash e_1 \; OR \; e_2 : Bool}$
$[AND_{EXP}]$	$\frac{E \vdash e_1 : Bool \qquad E \vdash e_2 : Bool}{E \vdash e_1 \; AND \; e_2 : Bool}$

Table G.21. Type rules for logical operators.

$[SELECTOR_{EXP}]$	$\frac{E \vdash all_1 : T_1, \ ... \ E \vdash all_n : T_n}{E \vdash id(all_1, \ ... \ , all_n) : Bool}$ $Where \ E(id) = (Selector, Bool, (T_1, \ ... \ T_n))$
$[CALCULATOR_{EXP}]$	$\frac{E \vdash e_1 : T_1, \ ... \ E \vdash e_n : T_n}{E \vdash id(e_1, \ ... \ , e_n) : T}$ $Where \ E(id) = (Calculator, T, (T_1, \ ... \ T_n))$
$[IDVAR_{EXP}]$	$\frac{E(id) = T}{E \vdash id : T}$

Table G.22. Type rules for ID expressions.

$[TFMETA_{EXP}]$	$E \vdash ThisFile(msv) : T$ $where \ T = MetaDataType(msv)$
$[TFISSELECTED_{EXP}]$	$E \vdash ThisFile.IsSelected : Bool$
$[TFHASHVALUE_{EXP}]$	$E \vdash ThisFile.HashValue : String$

Table G.23. Type rules for ThisFile expressions.

Statements

$[EMPTY_{STM}]$	$E \vdash \epsilon : ok$
$[COMP\_STMT_{STM}]$	$\frac{E \vdash S_1 : ok \qquad E \vdash S_2 : ok}{E \vdash S_1 S_2 : ok}$

Table G.24. Type rules for composite statement.

$[ASSIGN_{STM}]$	$\frac{E \vdash id : T \qquad E \vdash e : T}{E \vdash id = e : ok}$
$[IFELSE_{STM}]$	$\frac{E \vdash e : Bool \qquad E \vdash S_1 : ok \qquad E \vdash S_2 : ok}{E \vdash If(e)\ S_1\ Else\ S_2 : ok}$
$[IF_{STM}]$	$\frac{E \vdash e : Bool \qquad E \vdash S : ok}{E \vdash If(e)\ S : ok}$
$[FOREACHFILE_{STM}]$	$\frac{E \vdash S : ok}{E \vdash ForeachFile\ S : ok}$
$[BLOCK_{STM}]$	$\frac{E \vdash S : ok}{E \vdash \{\ S\ \} : ok}$

Table G.25. Type rules for statements.

$[VAR_{DEC}]$	$\frac{E[id \mapsto T] \vdash S : ok}{E \vdash T\ id;\ S : ok}$
$[VAR\_INIT_{DEC}]$	$\frac{E[id \mapsto T] \vdash S : ok \quad E \vdash e : T}{E \vdash T\ id = e;\ S : ok}$
$[SELECTOR_{DEC}]$	$\frac{E \vdash S_1 : ok \qquad E[id \mapsto (Selector, Bool, (T_1 \ldots T_n))] \vdash S_2 : ok}{E \vdash Selector\ id(T_1\ id_1, \ldots, T_n\ id_n)\{\ S_1\ \}\ S_2 : ok}$
$[CALC_{DEC}]$	$\frac{E \vdash S_1 : ok \qquad E[id \mapsto (Calculator, T, (T_1 \ldots T_n))] \vdash S_2 : ok}{E \vdash\ Calc < T > id(T_1\ id_1, \ldots, T_n\ id_n)\{\ S_1\ \}\ S_2 : ok}$
$[RULE_{DEC}]$	$\frac{\begin{array}{l}E \vdash S_1 : ok \ldots E \vdash\ S_n : ok \\ E \vdash e : Bool \quad E[id \mapsto (Rule,\ ok, ())] \vdash S : ok\end{array}}{E \vdash\ Rule\ id\{Select\ e; Do\ S_1, \ldots, S_n;\}\ S : ok}$

Table G.26. Type rules for declarations.

$[RUNRULES_{STM}]$	$\frac{E \vdash S : ok}{E \vdash RunRules\ \ id_1, \ldots, id_n;\ S : ok}$ $where\ E(id_1) = (Rule,\ ok, ()) \ldots E(id_n) = (Rule,\ ok, ())$

Table G.27. Type rules for RunRules statement.

$[DOMOVE_{STM}]$	$\frac{E \vdash e : Path}{E \vdash Move(e) : ok}$
$[DOCOPY_{STM}]$	$\frac{E \vdash e : Path}{E \vdash Copy(e) : ok}$
$[DORENAME_{STM}]$	$\frac{E \vdash e : String}{E \vdash Rename(e) : ok}$
$[DODELETE_{STM}]$	$Delete() : ok$

Table G.28. Type rules for Do calls.

PimplLib class diagram H

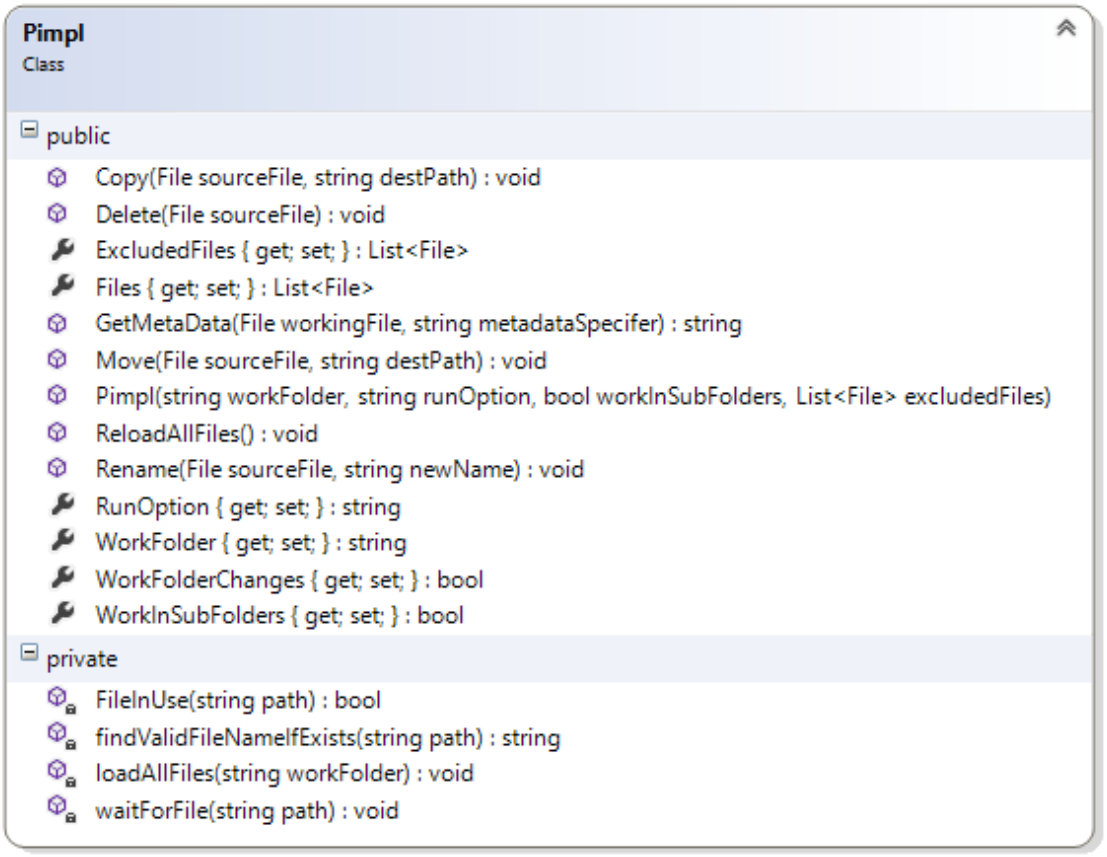

Pimpl
Class
public
Copy(File sourceFile, string destPath) : void
Delete(File sourceFile) : void
ExcludedFiles { get; set; } : List<File>
Files { get; set; } : List<File>
GetMetaData(File workingFile, string metadataSpecifer) : string
Move(File sourceFile, string destPath) : void
Pimpl(string workFolder, string runOption, bool workInSubFolders, List<File> excludedFiles)
ReloadAllFiles() : void
Rename(File sourceFile, string newName) : void
RunOption { get; set; } : string
WorkFolder { get; set; } : string
WorkFolderChanges { get; set; } : bool
WorkInSubFolders { get; set; } : bool
private
FileInUse(string path) : bool
findValidFileNameIfExists(string path) : string
loadAllFiles(string workFolder) : void
waitForFile(string path) : void

Figure H.1. Pimpl Runtime library class diagram. (1 af 2)

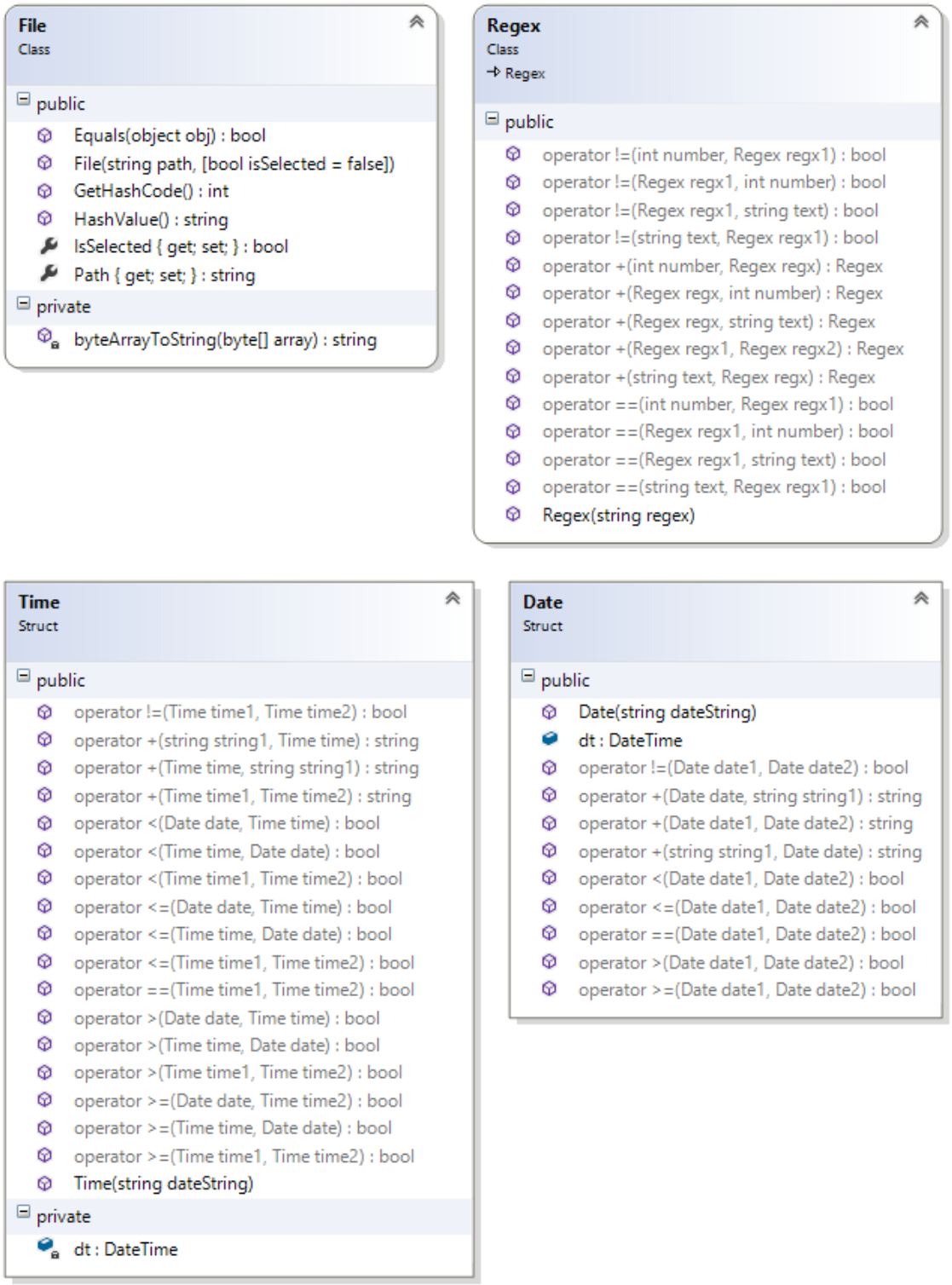

File
Class
public
Equals(object obj) : bool
File(string path, [bool isSelected = false])
GetHashCode() : int
HashValue() : string
IsSelected { get; set; } : bool
Path { get; set; } : string
private
byteArrayToString(byte[] array) : string
Regex
Class
Regex
public
operator !=(int number, Regex regx1) : bool
operator !=(Regex regx1, int number) : bool
operator !=(Regex regx1, string text) : bool
operator !=(string text, Regex regx1) : bool
operator +(int number, Regex regx) : Regex
operator +(Regex regx, int number) : Regex
operator +(Regex regx, string text) : Regex
operator +(Regex regx1, Regex regx2) : Regex
operator +(string text, Regex regx) : Regex
operator ==(int number, Regex regx1) : bool
operator ==(Regex regx1, int number) : bool
operator ==(Regex regx1, string text) : bool
operator ==(string text, Regex regx1) : bool
Regex(string regex)
Time
Struct
public
operator !=(Time time1, Time time2) : bool
operator +(string string1, Time time) : string
operator +(Time time, string string1) : string
operator +(Time time1, Time time2) : string
operator <(Date date, Time time) : bool
operator <(Time time, Date date) : bool
operator <(Time time1, Time time2) : bool
operator <=(Date date, Time time) : bool
operator <=(Time time, Date date) : bool
operator <=(Time time1, Time time2) : bool
operator ==(Time time1, Time time2) : bool
operator >(Date date, Time time) : bool
operator >(Time time, Date date) : bool
operator >(Time time1, Time time2) : bool
operator >=(Date date, Time time2) : bool
operator >=(Time time, Date date) : bool
operator >=(Time time1, Time time2) : bool
Time(string dateString)
private
dt : DateTime
Date
Struct
public
Date(string dateString)
dt : DateTime
operator !=(Date date1, Date date2) : bool
operator +(Date date, string string1) : string
operator +(Date date1, Date date2) : string
operator +(string string1, Date date) : string
operator <(Date date1, Date date2) : bool
operator <=(Date date1, Date date2) : bool
operator ==(Date date1, Date date2) : bool
operator >(Date date1, Date date2) : bool
operator >=(Date date1, Date date2) : bool

Figure H.2. Pimpl Runtime library class diagram. (2 af 2)

CD I

- PIMPL test programs
  - Delete duplicates
  - Organize pictures
  - Rename pictures
- PIMPL compiler
  - Compiler
  - Readme
- Source code
  - PIMPL compiler source code
  - PIMPL lib source code
- Existing solutions
  - Java test programs
- Design
  - Metadata mapping to PIMPL type
  - AST design
- PIMPL manual
  - User manual
  - PIMPL StandartLib
- Report